Magnifier: A Compositional Analysis Approach for Autonomous Traffic Control

Maryam Bagheri, Marjan Sirjani, Ehsan Khamespanah, Christel Baier, and Ali Movaghar

Abstract—Autonomous traffic control systems are large-scale systems with critical goals. Due to the dynamic nature of the surrounding world of these systems, assuring the satisfaction of their properties at runtime and in the presence of a change is important. A prominent approach to assure the correct behavior of these systems is verification at runtime, which has strict time and memory limitations. To tackle these limitations, we propose Magnifier, an iterative, incremental, and compositional verification approach that operates on a component-based model. The Magnifier idea is zooming on the component affected by a change, verifying the correctness of properties of interest of the system after adapting the component to the change, and then zooming out and tracing the change if it propagates. If the change propagates, all components affected by the change are adapted and are composed to form a new component. Magnifier repeats the same process for the new component. This iterative process terminates whenever the propagation of the change stops. In Magnifier, we use the Coordinated Adaptive Actor model (CoodAA) of traffic control systems. We present a formal semantics for CoodAA as a network of Timed Input-Output Automata (TIOAs). The change does not propagate if TIOAs of the adapted component and its environment are compatible. We implement our approach in Ptolemy II. The results of our experiments indicate that the proposed approach improves the verification time and the memory consumption compared to a non-compositional approach.

Index Terms—Self-adaptive Systems, Model@Runtime, Compositional Verification, Track-based Traffic Control Systems, Ptolemy II

1 Introduction

Many activities of the modern society are entirely managed by traffic control systems. These systems are large-scale, time and safety-critical systems that consist of numerous moving objects whose movements on a traveling space are adjusted and coordinated by controllers. The application domain of traffic control systems is not only limited to air traffic control systems or rail traffic control systems, but also includes more applications such as robotic systems, maritime transportation, smart hubs, intelligent factory lines, etc. The traffic in such systems can pass through pre-specified tracks, that based on the minimum safe distance between the moving objects, are partitioned into a set of sub-tracks. A system with this structural design is called a Track-based Traffic Control System (TTCS) [1].

Due to the dynamic nature of a TTCS and its surrounding world, a TTCS is vulnerable to failures, threatening human lives or causing intolerable costs. Autonomous response to context changes is a mechanism to prevent a failure in self-adaptive systems. Self-adaptive systems are able to adjust their structures and behaviors in response to changes. The controller in an autonomous TTCS uses the track-based design to safely and efficiently manage the traffic whenever an unpredicted change happens. For each change and its consequent adaptation, verifying the safety and quality of the system is necessary, which should be performed during the execution of the system. For performing the analysis and verification at runtime, an abstract model of the system and its environment, the so-called *model@runtime* [2], is generated, updated, and verified during the system execution.

In [3], we introduced the Coordinated Adaptive Actor model (CoodAA) for constructing and analyzing self-adaptive track-based traffic control systems. CoodAA is an actor-based [4], [5] approach augmented with coordination policies. In CoodAA each sub-track is modeled as an actor, the moving objects are considered as messages passed by the actors, and the controller is modeled as a coordinator. A TTCS is a large scale system partitioned into a set of control areas where each area has its own controller, so, a model of a TTCS can be intrinsically built as a set of components and is matched to CoodAA. The moving objects are sent and received at specified times through specified routes.

The coordinated adaptive actor model is designed based on the MAPE-K feedback loop [6] for self-adaptive systems. This control loop consists of the Monitor, Analyze, Plan, and Execute components. There is also a *Knowledge base* where the model@runtime is kept. The *Knowledge base* is updated by the *Monitor* component. The *Analyze* and *Plan* components are responsible for doing the analysis and providing adaptation plans when a change happens. The new plan is sent to the system through the *Execute* component.

In this paper, our focus is on the analysis that is performed for adaptation at runtime, and we propose the *Magnifier* idea. Magnifier uses an iterative and incremental process on a component-based model. When a change occurs Magnifier zooms-in on the affected component and checks if properties of interest still hold. If not, it adapts the component affected by the change by finding a new plan. Then, Magnifier checks if because of the new plan, the change is *propagated* through the model to other components, and it will continue the same process iteratively

M. Bagheri and A. Movaghar are with the Department of Computer Engineering, Sharif University of Technology, Iran. M. Sirjani is with the School of IDT, Mälardalen University, Sweden, and the School of Computer Science, Reykjavik University, Iceland. E. Khamespanah is with the School of Electrical and Computer Engineering, University of Tehran, Iran, and the School of Computer Science, Reykjavik University. C. Baier is with the department of Computer Science, Technical University of Dresden, Germany.

Manuscript received XX, 2020; revised XX, XXXX.

and incrementally. The idea is checking the effects of the change on the affected area and then in the least number of neighborhood components (and trying to contain it) instead of analysing the whole system for each change. The general idea of Magnifier is not specific for track-based traffic control systems and can be applied for any autonomous control system. But in our work, we focus on CoodAA and TTCSs, provide formal semantics and necessary theorems for compositional verification of TTCSs, and illustrate the results by implementing the approach.

In Magnifier, we use a compositional approach, we focus on the interface of each component which in CoodAA means the inputs and outputs of each component at a specified time. If the adapted component, according to the new plan, generates new outputs, or generates outputs with making new assumptions on its inputs it means that the effects of the change may propagate to the connected components. So, the connected components (or the so-called environment components) are adapted considering the new interface of the component. Then, Magnifier zooms-out and creates a new component by composing all components adapted to the change. The propagation of the change stops if the interface of the new (composite) component remains unchanged.

In this paper, we first present a compositional formal semantics for CoodAA as a network of Timed Input-Output Automata (TIOAs) [7]. Each component is represented by TIOAs of its constituent actors and its coordinator. We check the propagation of a change by checking the *compatibility* of TIOAs of the adapted component and TIOAs of its environment components. We call two (or more) TIOAs compatible if they do not reach a deadlock state in their parallel product.

To prove our incremental compositional approach, we adopt the compositional verification theorem of Clark et al. [8]. In [8], each component of the model is supplied with a correctness property. By composing a component with an abstraction of its environment components and verifying a property over the composition, the satisfaction of the property over the whole system is proved. Similar to [8], we use abstractions of the environment components. To reduce the state space, instead of TIOAs of the environment components, we only consider TIOAs of border actors that directly communicate with the adapted component. In contrast to [8], we do not use any logical formula to express the properties, since it is enough to check whether the adapted component interacts with its environment as expected (i.e. their compatibility).

Note that the verification of the *propagation* of a change is checking whether the interface of a component remains unchanged after adapting to a new plan. The verification is performed on the model@runtime that is a static snapshot of the system at the moment of the change.

To illustrate the applicability of our approach, we implement it in Ptolemy II [9]. Ptolemy II is an actor-oriented open-source modeling and simulation framework. A Ptolemy model consists of actors that communicate via message passing. The semantics of communications of the actors in Ptolemy is defined by models of computation, implemented in a set of predefined director components. Here, to provide assertion-based verification in Ptolemy II, we develop a Magnifier director. Our director generates the state space of the affected component, automatically extends its domain to include other components, and performs the reachability analysis over this extended domain. The results of our experiments for an example in the domain of air traffic control systems indicate a significant improvement in the verification time and the memory consumption.

Novelty and importance. Magnifier can be seen as a decentralized adaptation mechanism. Adaptation in a decentralized setting is a well-known challenge [10], [11]. It significantly improves the scalability and is a suitable option in hard real-time settings, when the reaction to a change should be performed in a negligible amount of time [11]. On the other hand, preserving global goals in a decentralized setting is difficult [11], as several components may need to reach a consensus about an adaptation policy to satisfy a global goal. Magnifier meets the global goals by first applying local adaptation to the component affected by a change. If it was not successful, it dynamically extends its adaptation (verification) domain to consider more components. The Magnifier approach relies upon the assumption that the environment components of a component are recognisable at the analysis time.

Contribution. CoodAA is introduced in [3] and its applicability in modeling TTCSs is shown by implementing a case study. In [1], CoodAA is explained using activity and sequence diagrams and the mapping between different applications of TTCSs and CoodAA is illustrated. We briefly presented the Magnifier idea as a future work in a short work-in-progress paper [12]. In the current paper, we present the formal foundation of CoodAA and Magnifier, and support the idea of effectiveness of Magnifier by an implementation of Magnifier in Ptolemy II and experimental results. The Summary of contributions are as follows:

- Formal compositional semantics of coordinated adaptive actor model as Timed Input-Output Automata (TIOAs)
- Compositional and incremental verification of model@runtime in CoodAA using Magnifier, and proof of correctness of the compositional approach
- Abstraction technique for environment components in Magnifier for reducing the state space
- Implementation of Magnifier as a director in Ptolemy II and supporting experimental results on an air traffic control system as an example to show the efficiency of Magnifier compared to a non-compositional approach

The rest of the paper is organized as follows. We provide a general overview of TTCSs in Section 2. We recall the definitions of a TIOA and the parallel composition of several TIOAs in Section 3. In Section 4, the formal compositional semantics of CoodAA is described in terms of TIOAs. Section 5 describes the details of the Magnifier approach. The implementation of Magnifier in Ptolemy II and the results of our experiments are shown in Section 6. We describe the related work in Section 7, and conclude the paper in Section 8.

2 Problem Definition and an Example

Track-based Traffic Control Systems (TTCSs), introduced in [1], are safety-critical systems that build on the idea of

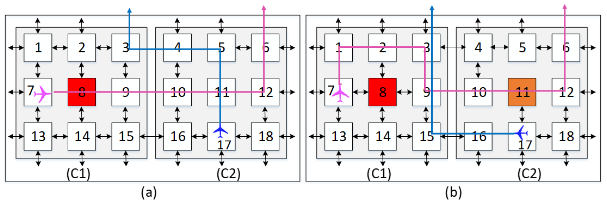


Fig. 1: A TTCS with 18 sub-tracks. The effect of the change in sub-track 8 is propagated to the component c_2. To avoid the collision in sub-track 11, the blue moving object is rerouted.

moving objects traveling across safe regions called sub-tracks. To reduce the risk of collision between moving objects in a TTCS, the traveling space is divided into a set of tracks. Based on the safe distance between two moving objects, each track is divided into smaller safe regions that are called sub-tracks. Each sub-track is a critical section that accommodates only one moving object in-transit. A large-scale TTCS is divided into a set of areas, while the traffic of each area is controlled by a centralized controller. The controller uses the track-based infrastructure to safely navigate the moving objects considering congestion and environmental changes. As explained in [1], the application domain of TTCSs ranges from Air Traffic Control Systems (ATCs), rail traffic control systems, maritime transportation, to centralized robotic systems and intelligent factory lines. For instance, ATC in the North Atlantic follows a track-based structure that is called an organized track system [13]. The North Atlantic organized track system consists of a set of nearly parallel tracks positioned in light of the prevailing winds to suit the traffic between Europe and North America.

In the real-world applications of TTCSs, each moving object has an initial traveling plan that is generated prior to the departure of the moving object from its source. A traveling plan consists of a route, time schedule decisions, and depending on the application, fuel, etc. The route of a moving object is a sequence of sub-tracks traveled by the moving object from its source to its destination. The time schedule decisions of a moving object consist of its departure time from its source, assumed arrival time at each sub-track in its route, and assumed arrival time at its destination. TTCSs are sensitive to unforeseen changes in their context. As a consequence of a dynamic environmental change, the traveling plans of moving objects may require to be modified. Therefore, following a change in the context, a sequence of changes might happen. For instance in an ATC, the aircraft flight plans are changed if a storm happens in a part of their flight routes. While changing traveling plans, several safety issues should be considered; i.e. loss of the separation between two moving objects should be avoided, and the remaining fuel should be checked. To avoid conflicts, changing the traveling plan of a moving object may result in changing the traveling plans of other moving objects. These changes can be propagated to the whole system. Besides the safety concerns, performance metrics such as arrival times of the moving objects at their destinations or sub-tracks in their routes are important. In a TTCS, the controller is in charge of coordinating the moving objects by rerouting/rescheduling them.

Example. An example of the change propagation is described for a TTCS as follows. Assume Fig. 1(a) and Fig. 1(b) show a TTCS with two control areas ($C1, C2$), where each area has nine sub-tracks. The traffic flows from the west to the east and vice versa. Each moving object of the eastbound traffic is able to travel towards a sub-track in the north, south, and east. The initial routes of the moving objects are shown in Fig. 1(a). The moving object with an unavailable sub-track in its route is rerouted and its new route is shown in Fig. 1(b). The red sub-track is an unavailable sub-track through which no moving object can travel. For instance, if a storm happens in a part of the airspace in an ATC, the aircraft cannot cross over the sub-tracks affected by the storm and are rerouted. Suppose that the traveling times of the moving objects through each sub-track are the same and are equal to one. The initial traveling planes of the purple and blue moving objects in Fig. 1(a) are $\{(0,7),(1,8),(2,9),(3,10),(4,11),(5,12),(6,6)\}$ and $\{(5,17),(6,11),(7,5),(8,4),(9,3)\}$, respectively. The first entry of each tuple shows the arrival time of the moving object at the sub-track mentioned in the second entry. For instance, two subsequent tuples $(0,7),(1,8)$ mean that the purple moving object arrives at sub-track 7 at time zero and arrives at sub-track 8 at time 1 (which is the same time that it exits sub-track 7).

Suppose that a change happens to sub-track 8 and it becomes unavailable. As a consequence, the traveling plan of the purple moving object is changed to $\{(0,7),(1,1),(2,2),(3,3),(4,9),(5,10),(6,11),(7,12),(8,6)$ $\}$, shown in Fig. 1(b). With the new plan, the purple moving object enters into sub-track 10 (next area) at time 5 instead of 3, and this way the change propagates from $C1$ to $C2$. Now, the purple moving object arrives at sub-track 11 at time 6. At this time, the blue moving object has to enter into sub-track 11 based on its initial traveling plan. To prevent the collision between two moving objects, the controller employs a rerouting algorithm (adaptation policy) and changes the plan of the blue moving object to $\{(5,17),(6,16),(7,15),(8,9),(9,3)\}$. As can be seen, by the occurrence of a change, e.g. a storm, a sequence of changes happens, e.g. rerouting a set of moving objects. This example also shows a situation in which the change circulates between two areas. Based on the new traveling plan obtained for the blue moving object, it enters into $C1$ at time 7 instead of 9, and this way the change propagates back to $C1$.

As a change in the context of a TTCS and its consequent adaptations in the system happen at runtime, the satisfaction of properties of interest should be check at runtime. The properties include: the moving objects have to arrive at their destinations at the pre-specified times, the collision of the moving objects should be avoided, the fuel of the moving objects should not be less than a threshold, and the system should be deadlock-free. These properties are checked by verification.

3 Background: Timed Input-Output Automata

In this section, we briefly recall the definitions of a TIOA, and the parallel product of several TIOAs. We also recall the definition of a deadlock state in a TIOA that is used to define the compatibility of two TIOAs in Section 5.

A timed automaton with a set of input actions and a set of output actions is called a TIOA. A TIOA with integer variables [14] is defined as follows.

Definition 3.1. *(TIOA) A Timed Input-Output Automaton is a tuple $TA = (Q, q_0, Var, Clk, Act_{in}, Act_{out}, T, I)$ where Q is a finite set of locations, $q_0 \in Q$ is the initial location, Var is the set of integer variables, Clk is a finite set of clocks, Act_{in} is a set of input actions, Act_{out} is a set of output actions, $T \in Q \times (\mathcal{B}(Clk) \cup \mathcal{B}(Var)) \times (Act_{in} \cup Act_{out} \cup \{\tau\}) \times 2^{Clk} \times 2^{Ass} \times Q$ is a set of edges, and I is an invariant-assignment function. Let $\# \in \{\leq, <, =, \geq, >\}$ and $c \in \mathbb{N}$. The sets of conjunctions of constraints of the form $x\#c$ or $x - y\#c$ for $x, y \in Clk$, and $v\#c$ or $v - w\#c$ for $v, w \in Var$ are respectively denoted by $\mathcal{B}(Clk)$ and $\mathcal{B}(Var)$. The set of all variable assignments is denoted by Ass. The function $I : Q \to \mathcal{B}(Clk)$ assigns invariants to locations.* □

Based on the above definition, the edge $e = (q, \psi, a, r, u, q') \in T$, besides action a, is labeled with a guard ψ, a sequence u of assignments, and a set r of clocks. Let $v_C, v'_C : Clk \to \mathbb{R}_{\geq 0}$ and $v_V, v'_V : Var \to \mathbb{Z}$ be clock and variable valuations, respectively. A state of the system modeled by a TIOA is in the form of (q, v_C, v_V). There is a discrete transition $(q, v_C, v_V) \xrightarrow{a} (q', v'_C, v'_V)$ for an edge $e = (q, \psi, a, r, u, q')$ such that v_C and v_V satisfy ψ, v'_C is reached by resetting the clocks in the set r to zero, and v'_V is obtained as a subset of variables are set to their new values in the assignment set u. The clocks and variables not mentioned in r and u remain unchanged. Furthermore, v'_C satisfies $I(q')$. The TIOA can stay in the location q as long as the invariant $I(q)$ is valid. Let for $x \in Clk$ and $d \in \mathbb{R}_{\geq 0}$, $(v_C + d)(x) = y + d$ iff $v_C(x) = y$. For each delay $d \in \mathbb{R}_{\geq 0}$ there is a timed transition $(q, v_C, v_V) \xrightarrow{d} (q, v_C + d, v_V)$ such that $v_C + d$ satisfies $I(q)$. A state of the system can be a deadlock state that, based on [15], is a state from which no outgoing discrete transition is enabled, even after letting time progress.

Definition 3.2. *(Deadlock State) A state s is a deadlock state if there is no delay $d \in \mathbb{R}_{\geq 0}$ and action $a \in (Act_{in} \cup Act_{out} \cup \{\tau\})$ such that $s \xrightarrow{d} s' \xrightarrow{a} s''$.* □

Consider the network $\mathcal{N} = \{TA_i | i = 1, \cdots, n\}$ of TIOAs, where TIOAs run in parallel and communicate through shared variables. Furthermore, TIOAs synchronize over time and shared common actions. We assume that when two edges (transitions) of two TIOAs synchronize over an action, their variables are updated by first executing the variable assignments of the output transition, and then by executing the variable assignments of the input transition. We also assume that the input transitions do not update the shared variables. Let $shared(\mathcal{N}) = \bigcup_{i,j=1_{j \neq i}}^{n} (Act_{in_i} \cap Act_{out_j})$ be the set of actions shared between two or more TIOAs in the network $\mathcal{N}$. In some cases, we use $a!$ and $a?$ instead of a to label an output and an input transition, respectively. For a subset $L = \{l_1, \cdots, l_m\}$, $l_i < l_{i+1}$, we define $\bigsqcup_{l \in L} u_l$ as a sequence of variable assignments $u_{l_1} \cdots u_{l_m}$. Based on [14], the parallel product of TIOAs in the network $\mathcal{N}$ is defined as follows.

Definition 3.3. *(Parallel Product) Let TIOAs $TA_i, i = 1, \cdots, n$, do not have shared output actions or shared clocks. The parallel product of $TA_1, \cdots, TA_n$, denoted by $TA_1 \otimes \cdots \otimes TA_n$, is $TA = (Q, q_0, Var, Clk, Act_{in}, Act_{out}, T, I)$, where*

$$Q = Q_1 \times \cdots \times Q_n, q_0 = (q_{0_1}, \cdots, q_{0_n}),$$

$$Var = \bigcup_{i=1}^{n} Var_i,\ Clk = \bigcup_{i=1}^{n} Clk_i,$$

$$Act_{in} = \bigcup_{i=1}^{n} Act_{in_i} \setminus shared(\mathcal{N}),$$

$$Act_{out} = \bigcup_{i=1}^{n} Act_{out_i} \setminus shared(\mathcal{N}),$$

$$I((q_1, \cdots, q_n)) = \bigwedge_{i=1}^{n} I_i(q_i),$$

and T is defined in the following way:

- *$((q_1, \cdots, q_n), \psi, a, r, u, (q'_1, \cdots, q'_n)) \in T$ iff there exists $i \in \{1, \cdots, n\}$ such that $a \in Act_{in_i} \cup Act_{out_i} \setminus shared(\mathcal{N})$, $(q_i, \psi, a, r, u, q'_i) \in T_i$, and for all $j \in \{1, \cdots, n\} \setminus \{i\}$, $q'_j = q_j$ holds;*
- *$((q_1, \cdots, q_n), \psi, \tau, r, u, (q'_1, \cdots, q'_n)) \in T$ iff there exists $a \in shared(\mathcal{N})$ and $i \in L$ for $L = \{k | (q_k, \psi_k, a, r_k, u_k, q'_k) \in T_k\}$ such that $(q_i, \psi_i, a!, r_i, u_i, q'_i) \in T_i$ and for all $j \in L \setminus \{i\}$, $(q_j, \psi_j, a?, r_j, u_j, q'_j) \in T_j$, and $\psi = \bigwedge_{k \in L} \psi_k$, $r = \bigcup_{k \in L} r_i$, $u = u_i \bigsqcup_{k \in L \setminus \{i\}} u_k$, and for all $k \in \{1, \cdots, n\} \setminus L$, $q'_j = q_j$ holds.* □

Note that the state of the system modeled by a network of TIOAs is obtained by clock values, values of all variables, and the locations of all TIOAs in the network. The same as the UPPAAL modeling language [16] we use functions to define invariants, guards, and variable assignments in the rest of this paper. Furthermore, we use different colors such as purple, green, light blue, and dark blue to respectively distinguish invariants, guards, synchronization actions, and clock reset and variable assignments in the figures related to TIOAs.

4 Formal Compositional Semantics of CoodAA

In this section, we first review the coordinated adaptive actor model. We then provide a formal compositional semantics for CoodAA in terms of TIOAs.

4.1 Background: Coordinated Adaptive Actor Model

We introduced the coordinated adaptive actor model in [3]. CoodAA consists of a set of coordination policies, which are described in a coordinator, and a set of actors. The actors can be categorized into a set of components. Each component has its own coordinator that describes the coordination policies relevant to the actors of the component. The structure of a component is shown in Fig. 2. A component can be a nested component; the actors of the component are categorized into a set of components. The coordinated adaptive

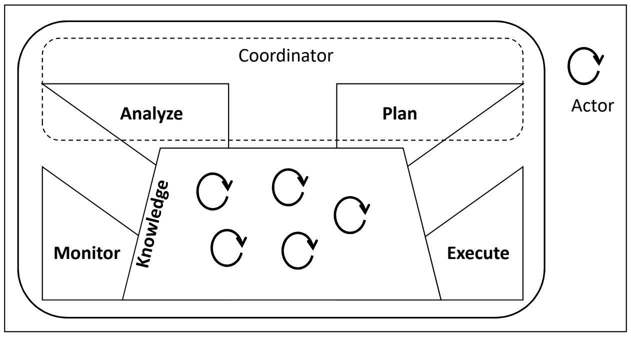


Fig. 2: The mapping between a component and the MAPE-K feedback loop. An actor-based model@runtime is kept in the *Knowledge base*.

actor model is designed based on the MAPE-K feedback loop to realize a self-adaptive system.

The managed system in a self-adaptive system is controlled by a feedback control loop. A well-known approach to realize a self-adaptive system is by means of the MAPE-K feedback control loop [6]. This loop consists of *Monitor*, *Analyze*, *Plan*, and *Execute* components together with the *Knowledge base*. The model@runtime, as an abstraction of the system and its environment, is kept in the *Knowledge base*. The *Monitor* component monitors the system and its environment, and updates the model@runtime. In the case of detecting a change, the *Analyze* component analyzes the model@runtime. The analysis results are given to the *Plan* component, and the *Plan* component makes an adaptation plan. The adaptation plan is applied to the model@runtime and the model@runtime is again analyzed. If the requirements of the system are satisfied, the adaptation plan is sent to the system through the *Execute* component. Otherwise, the *Plan* component makes another adaptation plan.

As shown in Fig. 2, each component of CoodAA (and as a consequence CoodAA itself) is mapped to a MAPE-K feedback loop. The actors of the component construct the model@runtime and the coordinator consists of the Analyze and Plan activities [3]. The coordinator is able to analyze the model@runtime by executing it to investigate the future behavior of the system. The coordinator dispatches messages among actors to execute the actor-based model@runtime.

CoodAA is aligned with the structure of track-based systems, each sub-track is modeled by an actor, the controller is modeled by a coordinator, and the moving objects are modeled as messages passing among the actors [1]. Each message carries information such as the identifier of a moving object, its traveling plan (route and time schedule decisions), speed, fuel, etc. The coordinator is able to adapt the system by rerouting/rescheduling the moving objects considering the congestion and environmental conditions. Upon occurring a change, the model@runtime is updated based on a snapshot taken from the system. Then, the coordinator obtains new routing/scheduling plans. The coordinator operates either in the regular phase or in the adaptation phase. In the regular phase, the coordinator dispatches the messages, and the routes/schedules are given in the messages passed to the actors. In the adaptation phase, the coordinator makes decisions to adapt the system. After the decision making, the new routes/schedules are passed to the actors while the coordinator moves to the regular phase. The messages are passed between the actors based on the plans given to the actors.

In [1], the coordinator is augmented with different rerouting/rescheduling algorithms. The coordinator is able to predict the behavior of the system through executing the model@runtime, measure several metrics using simulation, and based on the calculated results, select the best algorithm for rerouting/rescheduling purpose. The analysis in [1] is based on simulation not verification.

CoodAA is initially inspired from an adaptive actor-based framework proposed by Khakpour et al. in [17]. The framework is called PobSAM (Policy-based Self-Adaptive Model) and is an integration of algebraic formalisms and actor-based Rebeca models. A hierarchical extension of PobSAM is proposed by Khakpour et al. in [18]. The Reactive Object Language, Rebeca [19], is an actor-based [4], [5] modeling language. Rebeca is used for modeling and formal verification of concurrent and distributed systems. The model of computation in Rebeca is event-driven and the communication is asynchronous. Timed Rebeca [20], [21] extends Rebeca to model the timing features. In PobSAM there is no explicit notion of a coordinator, no timing constraint, and no focus on verification at runtime.

4.2 Summary of Definitions

In this section, the basic elements of CoodAA, including actors, components, coordinators, and channels are formally described. The summary of definitions and notations are provided in Table. 1. An actor has a variable *status* that shows the status of the actor being free, occupied or in an adaptation phase. It also has a variable to keep the plan (*movingPlan*); the plan specifies the direction and the time to send out the message. An actor has several input and output ports (modeling several directions that a moving object can arrive at or depart from a sub-track). Input and output ports are communication interfaces of the actor with other actors. An actor has a set of message handlers and each message handler corresponds to an input port of the actor. The actor can be informed about an adaptation or an environmental change by receiving a message over a special input port, the *change* port.

Definition 4.1. *(Actor) An actor, a_i, with the unique identifier i is defined as $(status_i, movingPlan_i, Mtds_i, change_i(args)\{body\}, P_{I_i}, P_{O_i})$, where $status_i$ having a value of $\{Free, Occupied, Error/Adapt\}$ denotes the state of the actor, $movingPlan_i$ stores the traveling plan of the moving object, $P_{I_i} = \{p_{I_{i,j}} | j = 1, \cdots, directions\} \cup \{change_i\}$ and $P_{O_i} = \{p_{O_{i,j}} | j = 1, \cdots, directions\}$ are respectively the sets of input and output ports of the actor, $Mtds_i = \{interact_{i,j}(transP)\{body\} | j = 1, \cdots, directions\}$ is the set of message handlers of the actor with the sequence of input arguments $transP = (objectId, travelPlan)$, $change_i(args)$ is a special message handler with the sequence of input arguments $args = (errorORadapt, adapted, error, errorOver, newPlan)$, and $body = stm^*$ as the body of a message handler is a sequence of statements.* □

- a_i is an actor with the unique identifier i, defined as $a_i = (status_i, movingPlan_i, Mtds, Change, P_{I_i}, P_{O_i})$
  - $status_i$ with a value of $\{Free, Occupied, Error/Adapt\}$ denotes the state of the actor
  - $movingPlan_i$ stores the traveling plan given to the actor by a message
  - $Mtds = interact_{i,j}(transP)\{body\}$ is the set of message handlers of the actor a_i, $transP = (objectId, travelPlan)$, $body$ is a sequence of statements, and $j = 1, \cdots, directions$
  - $Change = change_i(args)\{body\}$ is a special message handler where the sequence of its input arguments is $args = (errorORadapt, adapted, error, errorOver, newPlan)$
  - $P_{I_i} = \{p_{I_{i,j}} | j = 1, \cdots, directions\} \cup \{change_i\}$ is the set of input ports of the actor a_i
  - $P_{O_i} = \{p_{O_{i,j}} | j = 1, \cdots, directions\}$ is the set of output ports of the actor a_i
- ch_i is a channel with the unique identifier i, defined as $ch_i = (s_i, d_i)$
  - s_i is the source end of the channel ch_i
  - d_i is the sink (destination) end of the channel ch_i
- C_i is a component with the unique identifier i, defined as $C_i = (A_i, CH_i, B_i, F_i)$
  - $A_i = \|_{j \in I_{C_i}} a_j$ is a set of actors $\{a_j | j \in I_{C_j}\}$ concurrently executing, where I_{C_i} is the set of identifiers of the actors belonging to C_i
  - CH_i is the set of all channels of C_i
  - $F_i : A_i \to A_i$ is the coordination function of C_i
  - $B_i : P_i \rightharpoonup CH_i$ is the binding function of C_i, where $P_i = \bigcup_{a_j \in A_i} (P_{I_j} \cup P_{O_j})$ is a set of ports of all actors of C_i
- $C_k = C_i \parallel C_j$ is a composite component with the unique identifier k, defined as $C_k = (A_k, CH_k, B_k, F_k)$
  - $A_k = A_i \cup A_j$ is a set of actors concurrently executing
  - $CH_k = CH_i \cup CH_j \cup NewCH$ is the set of channels of C_k, where $NewCH$ is a set of channels to connect boundary output ports of a component to boundary input ports of the other component. The sets of boundary input and output ports of the component C_i are respectively defined as $P_{I_{C_i}} = \{p | a_l \in A_i \wedge p \in P_{I_l} \wedge \nexists ch \in CH_i, (p, ch) \in B_i\}$ and $P_{O_{C_i}} = \{p | a_l \in A_i \wedge p \in P_{O_l} \wedge \nexists ch \in CH_i, (p, ch) \in B_i\}$
  - $F_k : A_k \to A_k$ is the coordination function of C_k
  - $B_k = B_i \cup B_j \cup NewB$ is the binding function of C_k, where $NewB : P_k \rightharpoonup NewCH$ and P_k is the set of all boundary input and output ports of the components C_i and C_j.
- A coordinated adaptive actor model is a composite component, which is a component itself.

TABLE 1: Summery of definitions and notations for the coordinated adaptive actor model (CoodAA)

The *status* variable of the actor has the initial value of *Free*, and the *movingPlan* variable is null. The main computation of the actor is performed in its message handlers. A statement in the body of a message handler can be an assignment statement ($assign$), a conditional statement ($cond$), a send statement, or a delay statement, i.e. $stm ::= assign | cond | send(p, msg) | delay(t)$, where p is an output port, msg is a sequence of values, and t is a time variable. Definitions of assignment and conditional statements are like in regular programming languages. The actor can send a sequence of values as a message over an output port using the $send$ statement. The actor is also able to introduce a delay during the execution of its message handlers. The $delay(t)$ statement models the passage of t units of time for the actor, where t is derived from the plan of the actor.

Two actors are connected via a primitive medium that is called a channel. A channel has a source and a sink (destination) end. Each port of an actor can be connected to an end of at most one channel.

Definition 4.2. *(Channel) A channel, ch_i, with the unique identifier i is defined as (s_i, d_i), where s and d are respectively the source and the sink of the channel.* □

The source of a channel is connected to at most one output port and the sink of a channel is connected to at most one input port. The bindings between channels and ports of the actors are defined through the binding function of the component. A component is defined as follows.

Definition 4.3. *(Component) A component, C_i, with the unique identifier i is defined as $C_i = (A_i, CH_i, B_i, F_i,)$, where A_i is the set of internal actors of C_i, CH_i is the set of channels belonging to C_i, $F_i : A_i \to A_i$ is the coordination function of C_i, and $B_i : P_i \rightharpoonup CH_i$ is the binding function of C_i, where $P_i = \bigcup_{a_j \in A_i} (P_{I_j} \cup P_{O_j})$ is a set of ports of all actors of C_i.* □

The coordinator of a component is defined in the form of a function. In the rest of the paper, we use the terms coordinator and coordination function interchangeably. The coordination function of C_i is able to adapt the behaviors of actors by putting messages on the *change* ports of the actors. The binding function of C_i defines the topology of the component by connecting ports of actors to ends of channels.

Each component has sets of boundary input and output ports through which the component communicates with other components. These sets are defined based on the sets of input and output ports of the constituent actors of the component. The input port $p \in \bigcup_{a_l \in A_i} P_{I_l}$ is a boundary input port of the component C_i if p is not bound to any channels of the component C_i using the binding function B_i. Similarly, the output port $p \in \bigcup_{a_l \in A_i} P_{O_l}$ is a boundary output port of the component C_i if p is not bound to any channels of the component C_i using the binding function B_i. Therefore, the sets of boundary input and output ports of the component C_i are respectively defined as $P_{I_{C_i}} = \{p | p \in P_{I_l} \wedge a_l \in A_i \wedge \nexists ch \in CH_i, (p, ch) \in B_i\}$ and $P_{O_{C_i}} = \{p | p \in P_{O_l} \wedge a_l \in A_i \wedge \nexists ch \in CH_i, (p, ch) \in B_i\}$.

The composition of two (or more) components forms a composite component that is defined as follows.

Definition 4.4. *(Composite Component) Let $C_i = (A_i, CH_i, B_i, F_i)$ and $C_j = (A_j, CH_j, B_j, F_j)$ be two components. The composition of C_i and C_j, denoted by $C_i \parallel C_j$, results in the composite component $C_k = (A_k, CH_k, B_k, F_k)$ with the unique identifier k, where $A_k = A_i \cup A_j$, $CH_k = CH_i \cup CH_j \cup NewCH$ is the set of channels of C_k, $F_k : A_k \to A_k$ is the coordination function of C_k, and $B_k = B_i \cup B_j \cup NewB$ is the binding function of C_k. The set $NewCH$ is the set of channels that connect boundary output ports of a component to boundary input ports of the other component. The links between boundary ports and the new channels are defined using $NewB : P_k \rightharpoonup NewCH$, where P_k is the set of all boundary ports of the components C_i and C_j.* □

Based on Definition. 4.4, each composite component is itself a component. The coordinated adaptive actor model is a component composed of all components of the model.

4.3 Compositional Semantics of CoodAA

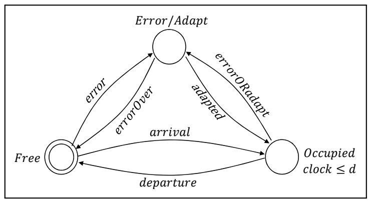
In this section, we present the compositional semantics of the coordinated adaptive actor model of a track-based system. Each component is presented in the form of a network of TIOAs of actors and a coordinator, which are shown in Fig. 3. Each actor is specified by a separate TIOA. There is no separate TIOA for a channel, since here we have zero-capacity channels, where each channel connects two ports and synchronises the communications of two actors. Fig. 3(a) shows an abstract view of an actor where changing the state between free, occupied and adaptation is clear. In Fig. 3(b) we show a more detailed view with more details on the interactions and transitions. We present an abstract TIOA of the coordinator in Fig. 3(c) which shows the operations of the coordinator on actors. We include an abstract view of the coordinator to present a complete formal semantics of CoodAA. This abstract view is enough for our discussions in this paper because our focus in this paper is on the verification of compatibility of components. In Magnifier, the coordinator analyzes and adapts the model@runtime, and the verification of compatibility of components that is performed in each iteration is on the model@runtime which is a snapshot of the system and only consists of the actors. The TIOA of the coordinator is not needed for that verification.

The TIOA of an actor, modeling a sub-track, without any details about the edges is shown in Fig. 3(a). This automaton has three locations that correspond to the values of the *status* variable of the actor. The *Free* location that is also the initial location represents that the sub-track is empty. The automaton moves from *Free* to *Occupied* whenever a moving object arrives at the sub-track. We suppose that the route of the moving object is updated over this transition. The route of a moving object is a sequence of sub-tracks traveled by the moving object. By passing a moving object from a sub-track, the first entry in the route of the moving object, referring to the current sub-track, is removed. The clock $clock$ represents the time elapsed since the arrival of the moving object at the sub-track. The traveling time of the moving object in Fig. 3(a) is denoted by d. The sub-track remains occupied during the traveling time of the moving object, formulated in the *Occupied* location. When a sub-track is occupied another moving object can not enter into it, because a sub-track is a critical section. The automaton moves from *Occupied* to *Free* whenever the moving object leaves the sub-track. A change can happen to a sub-track at any time, while the sub-track is empty or occupied. If the sub-track is empty and an adversarial event happens to the sub-track, the automaton moves from *Free* to *Error/Adapt*. The automaton moves back to the *Free* location the moment this adversarial event is removed. The automaton moves from *Occupied* to *Error/Adapt* if the sub-track is occupied and an adversarial event happens to the sub-track. The other case through which the automaton moves to *Error/Adapt* is when the sub-track is occupied and the traveling plan of the moving object should be adapted. The automaton moves back to the *Occupied* location whenever an adaptation decision in the form of rerouting/rescheduling for the traveling plan is made.

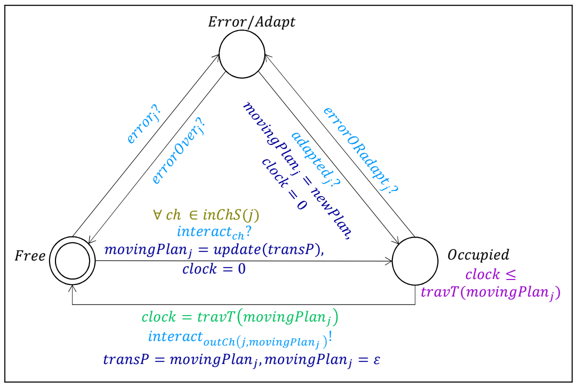
The TIOA of an actor a_j of the component C_i is shown in Fig. 3(b) with more details. This automaton has a *movingPlan* variable that corresponds to the *movingPlan* variable of the actor in Definition. 4.1. The automaton has a set of input actions $interact_{ch}$, where *ch* is the channel bound to an input port of the actor. These actions correspond to the interact handlers of the actor. Similarly, for each channel *ch* bound to an output port of the actor an output action $interact_{ch}$ is defined. Regarding the *change* port of the actor, the automaton has a set of input actions $errorORadapt_j$, $adapted_j$, $error_j$, $errorOver_j$, where each action has the same name as an input argument of the change handler. This automaton has access to the global variable *transP* that corresponds to the input argument of the interact handlers and is used to transfer a value between TIOAs of two actors. The global variable *newPlan* transfers a value from TIOA of the coordinator to TIOA of the actor. This variable corresponds to the *newPlan* argument of the change handler. The TIOA of an actor is defined as follows.

Definition 4.5. *(TIOA of an Actor) The TIOA associated with an actor a_j of C_i is $TA = (\{Free, Occupied, Error/Adapt\}, Free, Var, \{clock\}, Act_{in}, Act_{out}, T, I)$, where $Var = \{movingPlan_j\}$, $Act_{in} = \{interact_{ch} | ch \in inChS(j)\} \cup \{error_j, errorOver_j, errorORadapt_j, adapted_j\}$, $Act_{out} = \{interact_{ch} | ch \in outChS(j)\}$, $I(Occupied) = clock \leq travT(movingPlan_j)$, and T is defined as follows.*

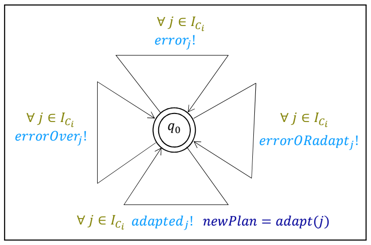
$$\forall ch \in inChS(j) : (Free, true, interact_{ch}, \{clock\}, \{movingPlan_j = update(transP)\}, Occupied) \quad \text{(Receive)}$$



(a) An abstract representation of TIOA of an actor a_j of the component C_i. The arrival of the moving object into the sub-track is modeled by the transition from *Free* to *Occupied*. The sub-track remains occupied during the traveling time of the moving object, formulated in the *Occupied* location, where $clock$ shows the time elapsed since the arrival of the moving object. The departure of the moving object from the sub-track is modeled by the transition from *Occupied* to *Free*. An actor goes into its *Error/Adapt* state when a change happens to the sub-track itself and/or the plan of the moving object needs an adaptation.

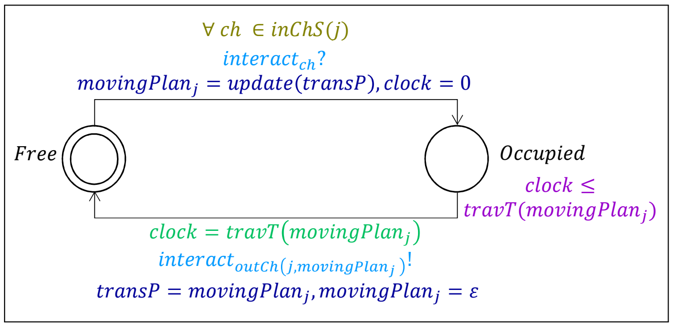

Error/Adapt
error$_j$?
errorOver$_j$?
movingPlan$_j$ = newPlan,
clock = 0
adapted$_j$?
errorORadapt$_j$?
∀ ch ∈ inChS(j)
interact$_{ch}$?
movingPlan$_j$ = update(transP),
clock = 0
Free
Occupied
clock ≤
travT(movingPlan$_j$)
clock = travT(movingPlan$_j$)
interact$_{outCh(j,movingPlan_j)}$!
transP = movingPlan$_j$, movingPlan$_j$ = ε

(b) The detailed model of TIOA of an actor that is shown in Fig. 3(a). The automaton moves from *Free* to *Occupied* by synchronizing with $interact_{ch}?$, where *ch* is an input channel of the actor. The variable *movingPlan* is the variable of the actor. The traveling plan in *movingPlan* is updated using the *update* function (the route of the moving object is updated by removing the first entry from the route). The function *travT* calculates the delay for the actor. The automaton moves from *Occupied* to *Free* by synchronizing with $interact_{ch}!$. The *transP* variable is used to transfer a message between TIOAs of two actors. The functions $inChS(j)$ returns the set of input channels of the actor. The function $outCh$ returns the output channel over which the message *movingPlan* is sent. The automaton moves to *Error/Adapt* by synchronizing with $error_j$ or $errorORadapt_j$, and moves back by synchronizing with $errorOver_j$ or $adapted_j$ of the coordination TIOA in Fig. 3(c).

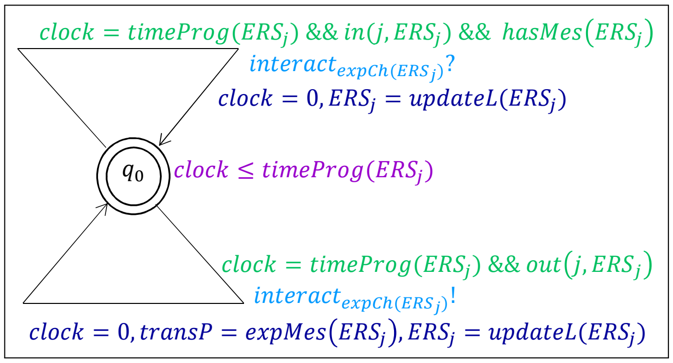


(c) The abstract representation of TIOA of the coordinator of the component C_i, where I_{C_i} is the set of identifiers of actors of C_i. The automaton changes the locations of the actors using a set of output actions, which are defined for each actor of the component C_i. The function *adapt* develops a new traveling plan that is sent to the actor a_j using the global variable *newPlan*.

Fig. 3: An abstract TIOA of an actor a_j of component C_i is shown in (a) and a more detailed view is shown in (b). An abstract representation of TIOA of the coordinator of the component C_i is shown in (c).


∀ ch ∈ inChS(j)
interact$_{ch}$?
movingPlan$_j$ = update(transP), clock = 0
Free
Occupied
clock ≤
travT(movingPlan$_j$)
clock = travT(movingPlan$_j$)
interact$_{outCh(j,movingPlan_j)}$!
transP = movingPlan$_j$, movingPlan$_j$ = ε

(a) The TIOA of an actor a_i of the component C_i in the model@runtime. This automaton is obtained from the automaton of Fig. 3(b) by removing the adaptation mode of the actor. This is because the model@runtime is analyzed after the coordinator applies its adaptation decision to the actors.


clock = timeProg(ERS$_j$) && in(j, ERS$_j$) && hasMes(ERS$_j$)
interact$_{expCh(ERS_j)}$?
clock = 0, ERS$_j$ = updateL(ERS$_j$)
q$_0$
clock ≤ timeProg(ERS$_j$)
clock = timeProg(ERS$_j$) && out(j, ERS$_j$)
interact$_{expCh(ERS_j)}$!
clock = 0, transP = expMes(ERS$_j$), ERS$_j$ = updateL(ERS$_j$)

(b) The TIOA of an augmented environment actor aa_j belonging to C_i. The augmented environment actor aa_j has a list *ERS* of messages that have to be sent or received by the actor at the pre-specified times and over the pre-specified channels. The actor stays at state q_0 until the time progresses up to a time at which aa_j sends or receives a message. The time required to be elapsed to send or receive a message is calculated by the function *timeProg*. This automaton synchronizes on the action $interact_{expCh(ERS_j)}$ to receive or send a message over an expected channel *expCh(ERS)*. Functions *in* and *out* determine if a message should be received or sent by aa_j, respectively. The top edge represents receiving, and it is only enabled if the message to be received matches the message that aa_j is expecting. This condition is checked by the function *hasMsg*. On the lower, edge aa_j sends a message, and using the function *expMes*, the first message in the list is added to *transP* which denotes passing the message by aa_j to the receiver actor. Finally, the function *updateL* removes the first entry from the *ERS* list.

Fig. 4: The TIOAs of an actor and an augmented environment actor in the model@runtime are respectively shown in (a) and (b). The automaton presented in (b) is described in Section. 5.

$$(Occupied, clock = travT(movingPlan_j), interact_{outCh(j,movingPlan_j)}, \emptyset, \{transP = movingPlan_j, movingPlan_j = \epsilon\}, Free) \quad \text{(Send)}$$

$$(Free, true, error_j, \emptyset, \emptyset, Error/Adapt) \quad \text{(Error)}$$

$$(Error/Adapt, true, errorOver_j, \emptyset, \emptyset, Free) \quad \text{(NoError)}$$

$$(Occupied, true, errorORadapt_j, \emptyset, \emptyset, Error/Adapt) \quad \text{(Adapt)}$$

$$(Error/Adapt, true, adapted_j, \{clock\}, \{movingPlan_j = newPlan\}, Occupied) \quad \text{(Adapted)}$$

□

In the following, we describe edges of TIOA of the actor a_j. For the sake of convenience, we call the channel bound to an input port of an actor an input channel of the actor. Similarly, we call the channel bound to an output port of an actor the output channel of the actor.

An actor is always waiting to receive a message over an input port. If a message is present, the actor receives the message and the message handler corresponding to that input port is triggered. By triggering $interact_{j,l}(transP)$ of the actor a_j on its input port $p_{j,l}$, the actor receives the message *transP* sent from another actor. When the message is received, the *status* variable of the actor is set to *Occupied*. The message *transP* includes the object id of the message ($objectId$) and the travel plan ($travelPlan$); *travelPlan* includes the traveling route of the moving object, its schedule, the amount of fuel, and its speed. This information is stored in the *movingPlan* variable of the actor. The actor updates the traveling plan (by removing the first entry in the route) before storing it in *movingPlan*. These operations are specified through the *Receive* edge that is defined for every input channel of the actor. The actor a_j of C_i receives a message from the port connected to the channel *ch* whenever TIOA of the actor is synchronized with the input action $interact_{ch}$. The auxiliary functions used over this edge are described as follows. Let AId be the set of all actor identifiers, CH be the set of all channels in the model, and Msg be the set of all messages in the form of (*objectId*, *travelPlan*). The function $inChS(j)$, where $inChS : AId \to 2^{CH}$, returns the set of all input channels of the actor a_j. The function $update : Msg \to Msg$ receives a message and returns a new message in which the traveling plan is updated.

The actor introduces a delay that is derived based on the traveling plan included in the *movingPlan* variable. This operation is formulated in the *Occupied* location of the automaton, where the function $travT : Msg \to \mathbb{R}_{\geq 0}$ receives a message and calculates the amount of the delay. After passing the delay time, the actor sends out the message. The output port of the actor over which the message is sent is also determined using the traveling plan. After sending the message, the *status* variable of the actor is set to *Free* and *movingPlan* is set to *null*. These operations are specified through the *Send* edge that is defined for every output channel of the actor. The function $outChS(j)$, where $outChS : AId \to 2^{CH}$, returns the set of all output channels of the actor a_j. The automaton of the actor is synchronized with the output action $interact_{outCh(j,movingPlan_j)}$ over this edge. The function $outCh(j, movingPlan_j)$, where $outCh : AId \times Msg \to CH$, returns the output channel of the actor a_j over which channel the message $movingPlan_j$ is sent. The message is transferred between two actors whenever their TIOAs are synchronized over an interact action. The message is delivered to the receiver actor using the *transP* variable.

The execution of an interact handler of the actor cannot be preempted by other interact handlers (the sub-track is a critical section), but the change handler has the highest priority for execution and can preempt interact handlers. By triggering the change handler, the message (*errorORadapt*, *adapted*, *error*, *errorOver*, *newPlan*) sent from the coordinator is received. The *status* variable of the actor is set to *Error/Adapt* under the following conditions. First, *error* is true and *status* is *Free*, second, *errorORadapt* is true and *status* is *Occupied*. The value of *status* changes from *Error/Adapt* to *Free* if *errorOver* is true. Furthermore, the value of *status* changes from *Error/Adapt* to *Occupied* if *adapted* is true. If the latter case holds, *movingPlan* variable of the actor is set to the new plan stored in *newPlan* of the message. These operations are specified through the *Error*, *NoError*, *Adapt*, and *Adapted* edges. The automaton respectively is synchronized with the input actions $error_j$ and $errorOver_j$ over the *Error* and *NoError* edges. Furthermore, the automaton respectively is synchronized with the input actions $errorOradapt_j$ and $adapted_j$ over the *Adapt* and *Adapted* edges.

An abstract TIOA for the coordinator is presented in Fig. 3(c). As can be seen, this automaton is synchronized with the automaton of an actor over the actions $error_j$, $errorOver_j$, $errorOradapt_j$, and $adapted_j$. This shows the role of the coordinator to update the state and/or the traveling plan of an actor.

4.3.1 TIOA of an Actor in the Model@runtime

The adaptation mechanism of coordinator obtains new plans for the actors. In Magnifier, the model@runtime is verified after an adaptation decision is applied to the model. An actor with *movingPlan*=null and *status=Error/Adapt* (an empty sub-track with an adversarial environmental condition) does not contribute in communications. Therefore, the *Error/Adapt* location is not needed in TIOA of an actor in the model@runtime. As shown in Fig. 4(a), we simplify TIOA of the actor a_j of the component C_i. Compared to Definition. 4.5, this simple TIOA does not contain the *Error/Adapt* location of the actor.

5 Verification of Model@runtime Using Magnifier

In this section, we develop a compositional approach to verify the system in the case of a change occurring and applying adaptation to components. We first provide insight into the Magnifier approach. We then present the definitions used to describe the approach. Finally, we formally explain the Magnifier approach and prove its correctness.

5.1 Overview

In this section, we present an overview of the Magnifier approach. We first informally explain the approach on track-based systems. When a track-based system is designed, initial traveling plans of the moving objects are selected in a way that no conflict happens between the moving objects, and the moving objects arrive at their destinations at the pre-specified times. In fact, the initial traveling plan of a moving object imposes constraints on the arrival of the moving object at each area of its route. When a change happens to an area, the moving objects traveling across the area are rerouted if there is an unavailable sub-track in their routes. This way the plan for the area is adapted. Note that the presence of a change (or its effect) in an area may last for a while, so any change in the plan must consider the possible future effects. Under the following conditions, the correctness properties of the system are satisfied and the change does not propagate:

1) The moving objects traveling across the area depart from it based on their initial traveling plans.
2) The moving objects intending to enter into the area at a future time, arrive at the area based on their initial traveling plans.
3) The moving objects entering into the area at a future time, depart from the area based on their initial traveling plans.

In the case of violating one of the above conditions, the change is propagated to the adjacent areas.

There is a set of properties for track-based systems that has to be satisfied. Collision of moving objects is avoided by design, a sub-track can only contain one moving object at one moment in time. Moving objects must arrive at their destinations at the pre-specified times. This property is checked by Magnifier. We also check that the system is deadlock-free. A deadlock in a track-based system happens whenever moving objects are stuck in a traffic blockage and cannot find an available route towards their destinations. If a moving object is stuck in some area of the traveling space, then it does not depart from the area. This causes propagation of the change through the whole system, and the object does not arrive at its destination on time. Deadlock can be caught by Magnifier. We also check if the fuel of moving objects go under a certain threshold. We use separate functions to detect a deadlock or running out of fuel and to stop the analysis.

The adaptation for an adjacent area is triggered whenever the change propagates into the area. This means that the traveling plans of the moving objects entering into the area or traveling across the area may need to be adapted. Therefore, in the case of propagating the change, all areas affected by the change are composed to form a new area. The traveling plans of the moving objects traveling across the new area are adapted. If the moving objects arrive at the new area and depart from it based on their initial traveling plans, the change propagation stops.

As explained informally for track-based systems, the Magnifier approach uses an iterative algorithm to assure the correctness of the system by involving the least number of the components in the analysis. The environment of a component is abstracted to the external messages that are sent to the component at the pre-specified times. When a change occurs, Magnifier zooms-in on the component affected by the change. The model@runtime corresponding to the component is updated based on the snapshot taken from the system at the change point. The model@runtime is adapted based on an adaptation policy defined in the coordinator. The adaptation contains the change if the adapted component can work with its environment by satisfying the constraints on their interactions. In fact, the propagation of the change stops if the adapted component is able to receive messages of its environment at the pre-specified times and over pre-specified ports, and guarantees to deliver messages to its environment at the pre-specified times and over pre-specified ports. Otherwise, the change is propagated to the environment. In this case, Magnifier zooms-out and concentrates on a new component resulted from composing all components affected by the change. The new component is adapted. The same procedure is repeated for the new component. In other words, the propagation of the change stops if the new component can work with its environment. Otherwise, Magnifier zooms-out and extends its verifying domain by composing the components affected by the change propagation and the previous ones.

One can argue that in a compositional approach we can check the change in one component and then check its propagation to the neighborhood components one by one. But a change may propagate back to the component which was the source of the change and develop a circular dependency. This situation is shown in the example of Section. 2. In Magnifier, by composing the components and forming a new component, all changes circulating between two components happen inside of the new component and their effects are considered.

5.2 Preliminary Definitions for Magnifier

In this section, we present the definitions on which the Magnifier approach relies. The notations and the summery of definitions are given in Table. 2. Let $C_{\mathcal{M}} = (A_{\mathcal{M}}, CH_{\mathcal{M}}, B_{\mathcal{M}}, F_{\mathcal{M}})$, composed of a set of components, be the coordinated adaptive actor model of a track-based system, where $A_{\mathcal{M}}$ is the set of actors, $CH_{\mathcal{M}}$ is the set of channels, $B_{\mathcal{M}}$ is the binding function, and $F_{\mathcal{M}}$ is the coordination function of $C_{\mathcal{M}}$. The component C_i of $C_{\mathcal{M}}$ models an area of the system, and interacts with a set of components called environment components of C_i. Let $P_{C_i} = \{p | a_l \in A_i \wedge p \in (P_{I_l} \vee P_{O_l}) \wedge \nexists ch \in CH_i \cdot (p, ch) \in B_i\}$ be the set of boundary ports of the component C_i. An environment component is defined as follows.

Definition 5.1. *(Environment Component) The component C_j is called an environment component of the component C_i if there exists a channel ch connecting a boundary port p_i of the component C_i to a boundary port p_j of the component C_j, i.e. $ch \in CH_{\mathcal{M}}, p_i \in P_{C_i}, p_j \in P_{C_j}, \{(p_i, ch), (p_j, ch)\} \subseteq B_{\mathcal{M}}$. The set of all environment components of C_i is denoted by $Env(C_i)$.* □

Using Definition. 5.1, we define an environment actor. An environment actor of a component is an actor whose input and output ports are bound to the input and output ports of the component by a set of channels (and hence directly sends or receives messages to/from the component).

- C_j is an environment component of C_i, $C_j \in Env(C_i)$, if $\exists ch \in CH_{\mathcal{M}} \cdot p_i \in P_{C_i}, p_j \in P_{C_j}, \{(p_i, ch), (p_j, ch)\} \subseteq B_{\mathcal{M}}$, where C_i and C_j are components of $C_{\mathcal{M}} = (A_{\mathcal{M}}, CH_{\mathcal{M}}, B_{\mathcal{M}}, F_{\mathcal{M}})$ and $P_{C_i} = \{p | a_l \in A_i \wedge p \in (P_{I_l} \vee P_{O_l}) \wedge \nexists ch \in CH_i \cdot (p, ch) \in B_i\}$ is the set of boundary ports of the component C_i. The set of all environment components of C_i is denoted by $Env(C_i)$
- a_k of C_j is an environment actor of C_i, $a_k \in C_j|_{C_i}$, iff $C_j \in Env(C_i) \wedge \exists ch \in CH_{\mathcal{M}}, p_j \in (P_{I_j} \vee P_{O_j}), p_i \in P_{C_i} \cdot \{(p_i, ch), (p_j, ch)\} \subseteq B_{\mathcal{M}}$. The set of actors of C_j that are environment actors of C_i is denoted by $C_j|_{C_i}$
- aa_k is the augmented environment actor, corresponding to the environment actor $a_k \in C_j|_{C_i}$, defined as $(ERS_k, init_k()\{body\}, Mtds_k, P_{I_k}, P_{O_k})$
  - ERS_k is an ordered list. Each entry of ERS_k is defined as $(transP, t, ch)$, where $transP = (objectId, travelPlan)$ is a message, t is a delay value, and ch is a channel identifier
  - $init_k()\{body\}$ is an initialisation method where *body* is a sequence of statements
  - $Mtds_k = interact_{k,l}(transP)\{body\}$ is the set of message handlers of aa_k where $l = 1, \cdots, Num$ and Num is the number of input ports of a_k through which a_k interacts with C_i
  - P_{I_k} is the set of input ports of aa_k. Each port is an input port of a_k through which a_k communicates with C_i
  - P_{O_k} is the set of output ports of aa_k. Each port is an output port of a_k through which a_k communicates with C_i
- $C_j \downarrow_{C_i}$ is the set of augmented environment actors of C_i where each actor of this set corresponds to an actor of $C_j|_{C_i}$
- $\mathcal{N}_{C_i}$ denotes the network of TIOAs of the component C_i
- $\mathcal{N}_{C_{a,i}}$ denotes the network of TIOAs of the adapted component C_i
- $\mathcal{N}_1 \bowtie \cdots \bowtie \mathcal{N}_n$ denotes that the networks $\mathcal{N}_1, \cdots, \mathcal{N}_n$ are compatible, where the parallel product of their TIOAs does not reach a deadlock state

TABLE 2: Summery of definitions and notations for the Magnifier approach

Definition 5.2. *(Environment Actor) Let $C_j \in Env(C_i)$ be an environment component of C_i. The actor a_j of C_j is an environment actor of C_i iff $\exists ch \in CH_{\mathcal{M}}, p_j \in (P_{I_j} \vee P_{O_j}), p_i \in P_{C_i} \cdot \{(p_i, ch), (p_j, ch)\} \subseteq B_{\mathcal{M}}$.* □

We use $C_j|_{C_i}$ to denote the set of actors of C_j that are environment actors of C_i. Suppose that $a_k \in A_j$ is an environment actor of the component C_i. We use $P_{I_{k,C_i}}$ and $P_{O_{k,C_i}}$ to respectively denote the sets of input and output ports of the actor a_k over which it communicates with the component C_i, i.e. $P_{I_{k,C_i}} = \{p \in P_{I_k} | \exists p_i \in P_{C_i}, ch \in CH_{\mathcal{M}} \cdot \{(p, ch), (p_i, ch)\} \subseteq B_{\mathcal{M}}\}$, $P_{O_{k,C_i}} = \{p \in P_{O_k} | \exists p_i \in P_{C_i}, ch \in CH_{\mathcal{M}} \cdot \{(p, ch), (p_i, ch)\} \subseteq B_{\mathcal{M}}\}$.

Abstraction of the environment. In order to abstract the environment of a component in Magnifier, corresponding to each environment actor an augmented environment actor is defined for the component. We augment all the significant information of an environment component to the augmented environment actor. An augmented environment actor has a list of expected receives and sends, called *ERS*, where each entry of the list contains a message, a delay value, and the identifier of a channel. Besides, this actor has a set of input ports, a set of output ports, an init method, and a set of message handlers, where each handler corresponds to an input port of the actor. The augmented environment actor corresponding to the environment actor a_k has $P_{I_{k,C_i}}$ and $P_{O_{k,C_i}}$ as the sets of its input and output ports, respectively.

Definition 5.3. *(Augmented Environment Actor) An augmented environment actor, aa_k, corresponding to the environment actor a_k, is defined as $(ERS_k, init_k()\{body\}, Mtds_k, P_{I_k}, P_{O_k})$, where ERS_k is an ordered list, $init_k$ is a method, $Mtds_k = \{interact_{k,l}(transP)\{body\} | l = 1, \cdots, Num\}$ is the set of message handlers of the actor with the sequence of input arguments $transP = (objectId, travelPlan)$, and $P_{I_k} = P_{I_{k,C_i}}$ and $P_{O_k} = P_{O_{k,C_i}}$ are the sets of input and output ports of the actor, respectively. The number of input ports is denoted by Num. Each entry of ERS_k is defined as $(transP, t, ch)$, where t is a delay value and ch is a channel identifier.* □

We use $C_j \downarrow_{C_i}$ to denote the set of augmented environment actors of C_i where each actor of this set corresponds to an actor of $C_j|_{C_i}$. We now define an interface component of a component. The interface components of a component (or visible parts of its environment) are sets of its augmented environment actors.

Definition 5.4. *(Interface Component) For each $C_j \in Env(C_i)$, $C_j \downarrow_{C_i}$ is called an interface component of the component C_i.* □

The definition of the interface component is inspired from the approach of [8], where it defines interface processes. For two processes P_1 and P_2, $P_1 \downarrow \Sigma_{P_2}$ is an interface process of P_2, where Σ_{P_2} is the set of symbols (i.e. atomic propositions) associated with P_2. The interface process $P_1 \downarrow \Sigma_{P_2}$ is the process P_1 in which all symbols that do not belong to Σ_{P_2} are hidden.

Finally, we define compatible TIOAs. In the Magnifier approach, two components can interact if their TIOAs are compatible.

Definition 5.5. *(Compatible TIOAs) Two or more TIOAs are compatible if the parallel product of them does not reach a deadlock state.* □

Our definition of compatibility is inspired from the approach of [22], in which two components (timed interfaces) are compatible if there is an environment to avoid the parallel product of the components from reaching an error state (the environment makes the components work together). In our approach, for checking the compatibility we do not consider any helpful environment. Note that a deadlock state in the product of two TIOAs is different from a deadlock in a track-based system. We use Definition. 5.5 to define the compatibility between networks of TIOAs. We call the networks $\mathcal{N}_1$ and $\mathcal{N}_2$ compatible if all TIOAs in $\mathcal{N}_1$ and $\mathcal{N}_2$ are compatible. Similarly, the networks $\mathcal{N}_1, \cdots, \mathcal{N}_n$ of TIOAs are compatible, denoted by $\mathcal{N}_1 \bowtie \cdots \bowtie \mathcal{N}_n$, if the networks are pairwise compatible.

5.3 Magnifier Approach

We first describe the Magnifier approach for two components. Prior to this, we use $\mathcal{N}_{C_i}$ to denote the network of TIOAs of the component C_i such that all TIOAs in $\mathcal{N}_{C_i}$ are compatible. We also use $\mathcal{N}_{C_{a,i}}$ to denote the network of TIOAs of the adapted component $C_{a,i}$ such that all TIOAs in $\mathcal{N}_{C_{a,i}}$ are compatible. Consider a system whose model consists of only two interacting components C_1 and C_2. To ensure that correctness properties of the system are satisfied, it is checked whether $\mathcal{N}_{C_1}$ and $\mathcal{N}_{C_2}$ are compatible in the absence of a change. None of the correctness properties are violated and there is a safe execution for the model if $\mathcal{N}_{C_1} \bowtie \mathcal{N}_{C_2}$. By detecting a change, the component affected by the change is adapted. Consequently, a new network for the adapted component is obtained. Suppose that a change in C_1 is detected. If $\mathcal{N}_{C_{a,1}} \bowtie \mathcal{N}_{C_2}$, the provided adaptation in C_1 does not result in a change propagation to its environment component (C_2) and no more adaptation is required. Otherwise, the change is propagated to C_2. This case shows that the provided adaptation changes the observable behavior of C_1, and C_2 has to be adapted to consider the new behavior of C_1.

Although the proposed approach works effectively for the small systems, checking the compatibility in a system with several components is an expensive process since the product of TIOAs of the adapted component and TIOAs of its environment components may result in a large state space. To reduce the state space in our analysis, we propose that instead of an environment component, its observable part to the component, which is the set of environment actors, is considered in the product.

As previously mentioned, the initial traveling plan of a moving object imposes constraints on the arrival of the moving object at each area of its route. The moving object arrives at each area of its route at a pre-specified time and from a pre-specified direction. Regarding the arrivals of the moving objects at a component, the environment actors send messages at the pre-specified times to the component, and regarding the departures of the moving objects from the component, the environment actors receive messages at the pre-specified times from the component. Therefore, the environment actors should abstract the environment of the component to the messages that are sent and received at the pre-specified times. To this end, in Magnifier an environment actor is replaced with an augmented environment actor that has the *ERS* list. An entry of *ERS* either specifies that the augmented environment actor expects to receive a message over an input port after an amount of time, or specifies that the actor intends to send a message over an output port after an amount of time. As the sub-track corresponding to an (augmented) environment actor is a critical section, *ERS* is an ordered list, where the first entry contains the message which should be sent or received by the actor first. Suppose that the augmented environment actor has received or sent the message of the first entry of *ERS* at time t. It will send or receive the message of the second entry at time $t + t'$ where t' is kept as a delay value in the second entry of *ERS*. The same argument is valid for the rest of the entries. As *ERS* in a model of a track-based system is calculated from the initial traveling plans of the moving objects, the schedules of the moving objects in *ERS* do not lead to any conflicts between the moving objects.

In the following, we define TIOA of an augmented environment actor aa_j that is used in the model@runtime. As shown in Fig. 4(b), this automaton has an *ERS* variable that corresponds to the *ERS* list of the actor. The automaton has a set of input actions $interact_{ch}$, where *ch* is a channel bound to an input port of the actor. Similarly, for each channel *ch* bound to an output port of the actor an output action $interact_{ch}$ is defined. This automaton has access to the global variable *transP* that corresponds to the input argument of the interact handlers and is used to transfer a value between TIOAs of an actor of CoodAA and the augmented environment actor.

Definition 5.6. *(TIOA of an Augmented Environment Actor) The TIOA associated with an augmented environment actor* aa_j *is* $TA = (\{q_0\}, q_0, Var, \{clock\}, Act_{in}, Act_{out}, T, I)$, *where* $Var = \{ERS_j\}$, $Act_{in} = \{interact_{ch} | ch \in inChS(j)\}$, $Act_{out} = \{interact_{ch} | ch \in outChS(j)\}$, $I(q_0) = clock \leq timeProg(ERS_j)$, *and* T *is defined as follows.*

$$(q_0, clock = timeProg(ERS_j) \wedge out(j, ERS_j) \\ , interact_{expCh(ERS_j)}, \emptyset, \{transP = expMes(ERS_j), \\ ERS_j = updateL(ERS_j)\}, q_0) \quad \text{(Send)}$$

$$(q_0, clock = timeProg(ERS_j) \wedge in(j, ERS_j) \wedge \\ hasMes(ERS_j), interact_{expCh(ERS_j)}, \{clock\}, \{ERS_j = \\ updateL(ERS_j)\}, q_0) \quad \text{(Receive)}$$

□

In the following, we describe edges of TIOA of aa_j. If a message is present for aa_j over an input port, aa_j receives the message, and the interact handler corresponding to that input port is triggered. In this case, the input arguments of the handler are not null. Besides the *init* method of the actor, an interact handler can trigger an interact handler by passing null values as input arguments. The *init* method of aa_j is executed whenever the actor is initialized. This method triggers an interact handler of aa_j. Let *e*=(*msg*, *t*, *ch*) be the first entry of the *ERS* list of the actor. By triggering an interact handler, the actor introduces a delay with the value of *t* if the input arguments of the handler are null. This operation of the actor is formulated in the q_0 location of the automaton shown in Fig. 4(b). The location q_0 ensures that the time progresses up to *t* at which the message *msg* is sent or received. Let En be the set of all entries of all *ERS* lists in the model. The function $timeProg(ERS)$, where $timeProg : 2^{En} \to \mathbb{R}_{\geq 0}$, returns t. After the time is progressed by t time units, the message *msg* should be sent or received.

If *ch* is bound to an input port of the actor, the actor expects to receive the message *msg*. In this case, the interact handler terminates and the actor waits to receive a message. If *ch* is bound to an output port of the actor the message *msg* should be sent over that port. After sending the message, *e* is removed from the *ERS* list, and the actor invokes the current executing interact handler with null values as arguments. This way, the same process for the first entry of *ERS* repeats. As shown in Fig. 4(b), the *Send* edge is enabled if $out(j, ERS_j)$, where $out : AId \times 2^{En} \to bool$, returns true. The function $out(j, ERS_j)$ determines if *ch* is bound to an output port of aa_j. The automaton of aa_j synchronizes over the action $interact_{expCh(ERS_j)}$ to send the message $expMes(ERS_j)$ over the channel $expCh(ERS_j)$. The function $expMes(ERS_j)$, where $expMes : 2^{En} \to Msg$, returns the message *msg*. The function $expCh(ERS_j)$, where $expCh : 2^{En} \to CH$, returns the channel *ch*. This edge uses $updateL(ERS_j)$, where $updateL : 2^{En} \to 2^{En}$, to remove *e* from the *ERS* list and returns the rest of the list.

By triggering an interact handler, if the input arguments of the handler are not null, the actor checks whether it has received the message *msg* over the input port bound to *ch*. An error is thrown if this condition does not hold. Otherwise, *e* is removed from *ERS*, and the actor invokes the current executing interact handler. This way, the same operation for the first entry of *ERS* repeats. As shown in Fig. 4(b), the *Receive* edge is enabled if the functions *in* and *hasMsg* return true. The function $in(j, ERS_j)$, where $in : AId \times 2^{En} \to bool$, returns true if *ch* is bound to an input port of the actor. The function $hasMes(ERS_j)$, where $hasMes : 2^{En} \to bool$, returns true if the message to be received is equal to *msg*. The automaton synchronizes on $interact_{expCh(ERS_j)}$ to receive *msg* over the channel *ch*, which is determined by $expCh(ERS_j)$. In the case of an error is thrown, TIOAs of aa_j and the sender actor reach a deadlock state.

In Magnifier, an interface component is described by a network of TIOAs where each TIOA models an augmented environment actor. Suppose that a change happens to a component, and the component is adapted to the change. If the network of TIOAs of the adapted component and the networks of TIOAs of its interface components are compatible, the change does not propagate to the environment components. Otherwise, the change is propagated to the environment components. Since each component has a well-defined interface, we are able to focus on a component and define an interface component for each one of its environment components. This way, we are able to find the direction where the change is propagated and to find the components affected by the change propagation. The change propagates from the component C_i to the component $C_j \in Env(C_i)$ if there is an actor of $C_j \downarrow_{C_i}$ such that no transition is performed in the location q_0 of its TIOA (Definition. 5.6) even after letting time progress. This shows a deadlock in the product of TIOAs of the component C_i and the interface component $C_j \downarrow_{C_i}$, and means that this augmented environment actor is not able to either send a message over

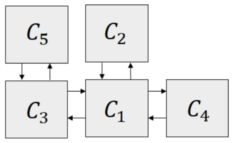


Fig. 5: A model consisting of 5 interactive components

a pre-determined port at a pre-specified time, or receive an expected message from a pre-determined port at a pre-specified time. In our approach, by propagating the change, all components affected by the change are composed to create a new component that is adapted (please note that an actor follows the semantics defined in Fig. 4(b) whenever it belongs to an interface component, but all actors have the semantics shown in Fig. 4(a) whenever two components are composed to make a new component). It is then checked whether the network of TIOAs of the new component and networks of TIOAs of its interface components are compatible.

Example. For a better understanding of the approach, consider the following example. A model with five components is shown in Fig. 5. The interactions between the components are denoted by arrows. Suppose that a change in the component C_1 of Fig. 5 is detected and this component (its model@runtime) is adapted. If $\mathcal{N}_{C_{a,1}} \bowtie \mathcal{N}_{C_2 \downarrow_{C_1}} \bowtie \mathcal{N}_{C_3 \downarrow_{C_1}} \bowtie \mathcal{N}_{C_4 \downarrow_{C_1}}$ does not hold, the change propagates into some of the environment components. Let the change propagates to the components C_2 and C_3. It means that C_1 with its current adaptation is not able to either receive messages from $C_i, i = 2, 3$, or send messages to C_i at the pre-specified times. Consequently, the adaptation for C_2 and C_3 is triggered. The components C_1, C_2, and C_3 are adapted and are composed to provide the new component $C_{1,2,3}$. If $\mathcal{N}_{C_5 \downarrow_{C_3}} \bowtie \mathcal{N}_{C_4 \downarrow_{C_1}} \bowtie \mathcal{N}_{C_{1,2,3}}$ holds, the change propagation stops, and the change is not propagated further than C_1, C_2, and C_3.

5.4 Correctness of the Proposed Approach

In the previous section, we defined interface components of an adapted component, where each interface component is an abstraction of an environment component. In the absence of a change, the correctness properties of the system are preserved as each component of the system preserves a set of local correctness properties, i.e. each component receives and sends messages at the pre-specified times and over the pre-specified ports. In the presence of a change, the correctness properties are satisfied if the adapted component can work with its environment, i.e. the adapted component and its environment satisfy their input and output assumptions. This is where the networks of TIOAs of the adapted component and its interface components are compatible. In this section, the correctness of the proposed approach is proved in Theorem. 5.1. This theorem explains that reducing the environment components to the interface components is correct.

Theorem 5.1. *The networks of TIOAs of the adapted component and its interface components are compatible if and only if the networks of TIOAs of the adapted component and its environment components are compatible.*

Proof. "if": By contradiction. Suppose that the networks of TIOAs of the adapted component C_i and its environment components are compatible, but $\mathcal{N}_{C_i} \bowtie \mathcal{N}_{C_{j_1}\downarrow_{C_i}} \bowtie \cdots \bowtie \mathcal{N}_{C_{j_n}\downarrow_{C_i}}$, where $C_{j_k} \in Env(C_i)$ and $k = 1, \cdots, |Env(C_i)|$, does not hold. It means that there exists an interface component $C_{j_l} \downarrow_{C_i}, C_{j_l} \in Env(C_i)$, and an augmented environment actor $aa_j \in C_{j_l} \downarrow_{C_i}$ such that this actor is not able to either receive an expected message from an expected port at a pre-specified time or send a message over a pre-specified port at a pre-specified time. Let aa_j corresponds to the environment actor a_j. However, the actor a_j belongs to C_{j_l}, e.g. $a_j \in A_{j_l}$. This means that $\mathcal{N}_{C_i} \bowtie \mathcal{N}_{C_{j_1}} \bowtie \cdots \bowtie \mathcal{N}_{C_{j_n}}$ does not hold, which contradicts the assumption.

"only if": By contradiction. Let the networks of TIOAs of the adapted component C_i and its interface components be compatible, but $\mathcal{N}_{C_i} \bowtie \mathcal{N}_{C_{j_1}} \bowtie \cdots \bowtie \mathcal{N}_{C_{j_n}}$, where $C_{j_k} \in Env(C_i)$ and $k = 1, \cdots, |Env(C_i)|$, does not hold. We assumed that the adaptation results in a new network of compatible TIOAs for the component C_i. Furthermore, as each component $C_j \in Env(C_i)$ is not yet affected by a change, all TIOAs in $\mathcal{N}_{C_j}$ are compatible. Therefore, there exists $C_j \in Env(C_i)$ and an environment actor $a_j \in A_j$ such that this actor is not able to either receive an expected message from an expected port at a pre-specified time or send a message over a pre-specified port at a pre-specified time. However, this actor corresponds to an actor of $C_j \downarrow_{C_i}$. This means that $\mathcal{N}_{C_i} \bowtie \mathcal{N}_{C_{j_1}\downarrow_{C_i}} \bowtie \cdots \bowtie \mathcal{N}_{C_{j_n}\downarrow_{C_i}}$ does not hold, which contradicts the assumption. $\square$

6 Implementing and Evaluating Magnifier Using Ptolemy II

In this section, we briefly describe the implementation of Magnifier for an ATC case study with several control areas in Ptolemy II [9] as a proof of concept for effectiveness and efficiency of the work. The main reason for using Ptolemy II instead of UPPAAL is that Ptolemy enables us to automate the iterative and incremental process of Magnifier using its so called *director*. The change propagated through the system can be automatically traced, and the verification scope can be extended to bring more components into the analysis. Here, we first give a background on Ptolemy II. We then describe our implementation and compare the time consumption and the memory consumption between the compositional and non-compositional approaches. The Ptolemy model and implementations of the provided algorithms in this section are available online[1].

6.1 Background: Ptolemy Framework

Ptolemy II [9] is an actor-based modeling and simulation framework that provides different models of computation with fully deterministic semantics. A model of computation defines the semantics of interactions of actors, and is implemented in a Ptolemy director. In [1], we developed a Ptolemy template based on the coordinated adaptive actor model to model and analyze self-adaptive TTCSs. In this template, each sub-track is modeled by a Ptolemy actor, and the moving objects are considered as messages passing among the actors. The pathways between the sub-tracks are modeled by interconnections between the actors. The Ptolemy actors are connected through channels. An actor can read a message form an input channel and send a message over an output channel. Furthermore, the controller (coordinator) is modeled by a Ptolemy director. In [1], we developed a director by extending the Discrete Event (DE) director. DE is one of the most commonly used models of computation in Ptolemy.

The DE director has a buffer of events. Each message communicated in the model is packaged in an event. An event besides a message has a time tag and a reference to the receiver actor in the communication. In fact, instead of direct message passing among the actors, upon sending a message, an event is created and is placed into the internal buffer of the director. The director keeps the model time and stamps the event with the model time at which the message has been sent. An actor can execute the *fireAt* instruction to ask the director to trigger the actor at a future time. In this case, an event with an empty message is generated for the director. The director labels the event with the requested future time. All the events tagged with the value of the current model time are enabled events. The director takes an enabled event from its buffer, triggers the actor referred to by the event, and delivers the message to the actor. To choose between a set of enabled events, a deterministic policy (the so-called topological sort) is used. If there is no enabled event, the model time progresses to reach the time of the event with the smallest time tag.

6.2 Ptolemy Implementation of Magnifier

In [1], we implemented CoodAA and its MAPE-K architecture by extending the DE director in Ptolemy II. We modeled ATC and railroad examples, and we checked different rerouting plans in different situations. The difference between the examples is in the number of ports of actors and the topology of their bindings. Our analysis in [1] was based on the simulation engine provided by Ptolemy II. In this paper, we use the Ptolemy II template proposed in [1], and extend the DE director to develop the Magnifier director that supports formal verification. In each iteration, after replanning, Magnifier builds the state space to check the compatibility of components.

Our implementation is faithful to the TIOA semantics of Section 4. As described in Section 6.1, the Ptolemy director keeps a buffer of events and therefore it can adapt the network by manipulating the messages encapsulated in the events. The Magnifier director mimics the non-deterministic selection of actors for being executed. From a set of enabled events, the Magnifier director takes one of them non-deterministically. Finally, if there is no enabled event (i.e. with the time tag equal to the current model time), the model time progresses. This non-determinism is shown in Fig. 4(a) since from a set of enabled edges in a network of TIOAs (i.e. multiple actors), an edge is selected non-deterministically.

In our implementation, each actor (sub-track) is modeled by a Ptolemy actor. An actor is first triggered to receive the message corresponding to a moving object. Then, the actor executes the fireAt instruction and asks the director to trigger it again at a future time to send its message out.

1. http://www.ce.sharif.ir/~mbagheri/MagImp.zip

This way the traveling time of the moving object through the sub-track is modeled. In the proposed semantics in Section 4, there is no centralized buffer as messages are directly delivered to the actors at the same model time. In our implementation, like in our semantics, the events created per sending a message are processed at the same model time at which the message has been sent. The global model time in Ptolemy mimics the synchronous progress of time of clocks in a network of TIOAs.

The magnifier director provides the assertion-based verification. It generates the state space of a given component, and performs the reachability analysis. For the sake of simplicity, we assume that all the coordinators of all components (the ATC controllers of all areas) have the same adaptation policy (rerouting algorithm). This way, we have only one coordinator (instead of a nested model and multiple coordinators). The Magnifier director generates the state space of the model of an ATC example with several components, where the components are composed to create a new component. The rerouting algorithm and the algorithm given in the following section are implemented in the director. It is notable that designing the rerouting algorithm is not the concern of this paper.

6.2.1 Generating the State Space

Here, we explain the algorithm to generate the state space. We also present the pseudocode of this algorithm.

Algorithm. The algorithm to generate the state space of a component is shown in Algorithm. 1. Let the initial state of the component be a timed state. We call a state a timed state if a time transition is enabled at the state. The algorithm uses a queue to store the timed states (line 3). It dequeues a timed state (line 5), and after progressing the time (line 6), uses Depth-First Search (DFS) to generate all the traces starting with that state and ending with the new timed states (line 8). The algorithm terminates whenever no new timed state is generated (lines 9-12). Otherwise, the generated timed states are added to the end of the queue (line 13). The function *timeProg* progresses the time to the smallest time of the events stored in the buffer of the director in state s, and the function *deQueue* removes and returns the first state of the queue.

The main computation of the algorithm is performed by the function *depthFS*, presented in Algorithm. 2. Let *buffer(s)* denotes the buffer of the events kept in the director in state s. The function *depthFS* takes an event (line 11), and using the function *trigger*, triggers the actor referred to by the event to generate the next state (line 12). Compared to state s, the buffer of the director in the next state stores the events that are possibly created by triggering the actor. Furthermore, the taken event is removed from the buffer in the next state. The state s has several outgoing transitions (resp. several next states) if several actors can be triggered at the state. The next state is returned if it is a timed state (lines 7-10). Otherwise, the actors which can be triggered in the next state are triggered. The algorithm to generate the state space terminates whenever one of the following conditions is fulfilled: all the moving objects supposed to travel through the component depart from it (reach their destinations), a disaster happens (i.e. the fuel of a moving object is zero), and the analysis time passes a threshold. These conditions are checked using the *terminate* function over a state (lines 3-6).

We use this algorithm to also generate the state space of a set of components in the non-compositional approach. It is notable that the state space of the system does not have a Zeno behavior, since the travel of every moving object across a sub-track takes time. The minimum progress of the time in our model is assumed one unit.

Algorithm 1: Algorithm to generate the state space

```
   Input: $s_0$ as the initial state of the component
   Output: stateSpace that is the state space
1  begin
2      $stateSpace \leftarrow \{s_0\}$
3      $queue \leftarrow \{s_0\}$
4      while $queue \neq \emptyset$ do
5          $s \leftarrow deQueue(queue)$
6          $s' \leftarrow timeProg(s)$
7          $stateSpace \leftarrow stateSpace \cup \{s'\}$
8          $states \leftarrow depthFS(s')$
9          if $states = \emptyset$ then
11             return $stateSpace$
12         end
13         $queue \leftarrow \langle queue | depthFS(s') \rangle$
14     end
16     return $stateSpace$
17 end
```

Algorithm 2: depthFS

```
   Input: $s$ as a state
   Output: timedStates as a set of timed states
1  begin
2      $timedStates \leftarrow \emptyset$
3      if $terminate(s)$ then
5          return $\emptyset$
6      end
7      if $timeStat(s)$ then
9          return $\{s\}$
10     end
11     foreach $e \in buffer(s)$ do
12         $s_1 \leftarrow trigger(s, e)$
13         $stateSpace \leftarrow stateSpace \cup \{s_1\}$
14         if $depthFS(s_1) = \emptyset$ then
16             return $\emptyset$
17         end
18         $timedStates \leftarrow timedStates \cup depthFS(s_1)$
19     end
21     return $timeStates$
22 end
```

Composition. Assume that a storm happens at time t, and the component C_1 is affected by the storm. Also, assume that flight plans of the aircraft are not necessarily the initial flight plans and have been adjusted based on the data monitored from the system. We use flight plans of the aircraft to extract the state of the system, describing positions of the aircraft in the traffic network at time t. We then set the initial state of C_1 to the state of the system.

Similarly, we use flight plans of the aircraft to obtain states of environment actors of C_1 by identifying the aircraft entering into C_1 in the future. These environment actors will directly send messages to C_1 at times t', $t' \geq t$, and expect to receive messages from C_1 at times t', $t' \geq t$. To generate the state space, we first compose C_1 with its environment actors to create a new component. We then use the above algorithm to generate the state space of the new component. If the composition of the state spaces of C_1 and the environment actors does not reach a deadlock state, the state space of the new component is successfully generated. Suppose the case in which the composition reaches a deadlock state. In this case, assume that an environment actor belonging to the environment component C_2 is not able to send its message at the pre-specified time $t'', t'' \geq t$, to C_1. This means that the change is propagated from C_1 to C_2 at time t''. We repeat the same procedure as C_1 to set the state of C_2 to the state of the real system at time t''. We also set states of those environment actors that send messages to C_2 at the times greater than t''. Therefore, from t'' on, we have a component, composed of C_1, C_2, and several environment actors, whose state space is generated. This procedure terminates whenever the algorithm reaches a state in which all moving objects supposed to travel across the new component depart from it at their pre-specified times. In other words, the model has a trace during which all messages are received from the new component at the pre-specified times.

6.3 Experimental Setting

To compare the compositional and non-compositional approaches, we focused on an ATC example with a $n \times n$ mesh map, where the location of each sub-track is shown by the pair (x, y) in the mesh. We also considered $2 \times (n - 1)$ source airports (each one is connected to a sub-track whose location is the pair $(0, i)$ or $(i, 0)$, $0 \leq i < n$), and $2 \times (n - 1)$ destination airports (each one is connected to a sub-track whose location is the pair $(n - 1, i)$ or $(i, n - 1)$). We developed an algorithm to generate the initial flight plans of m aircraft, and an algorithm (an adaptation policy) to reroute the aircraft as follows.

ALG1: Generating the initial plans. This algorithm randomly generates the source (x_s, y_s), the destination (x_d, y_d), and a departure time from the source airport for each aircraft. The departure times follow an exponential distribution with the parameter λ. The time difference between two subsequent departures from a source airport should not be less than the flight time FD, which shows the traveling time of an aircraft across a sub-track. The aircraft A can travel through the sub-track with the location (x, y) if A has no time conflict with the aircraft B, which is also supposed to travel across (x, y). Similar to the XY routing algorithm [23], ALG1 attempts to find a route from (x_s, y_s) to (x_d, y_d) by first traversing the X dimension and then traversing the Y dimension of the mesh. ALG1 switches its traversing direction from X to Y whenever the aircraft has a time conflict with another aircraft along the X dimension. ALG1 backtracks if it can move across none of the dimensions from the location (x, y). It then moves across the Y dimension. These procedure continues until a route is discovered. ALG1 does not guarantee to find the efficient (e.g. shortest) route.

ALG2: Rerouting algorithm. Assume that the aircraft is going to leave the location (x_0, y_0) and the rest of its route is $[(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)]$. Also, assume that the sub-track T with the location (x_1, y_1) is unavailable, and the moving object is not able to travel through it. In the ATC example, a sub-track is unavailable if it is stormy or is occupied by another aircraft. The algorithm finds a neighbor of (x_0, y_0), e.g. the sub-track T', that is available and neither of its x and y is equal to T. Then, the algorithm tries to find a route from T' in several steps. At the first step, the algorithm tries to find a route with the length 2 from T' to (x_2, y_2). If there is no such route, it attempts to find a route with the length 3 from T' to (x_3, y_3), and so on. If a route from T' to (x_i, y_i), $2 \leq i \leq n$, is found, the route is concatenated with the current route of the aircraft from (x_{i+1}, y_{i+1}) to (x_n, y_n). In fact, ALG2 attempts to find a route with the length equal to the length of the initial route. However, if (x, y) does not have an available neighbor, or a route with the same length as the initial route is not found, the algorithm finds a route from a neighbor (the neighbor in this case might be occupied). Then, the aircraft will stay one more unit of time in (x, y), and will fly based on its new route. If no route is found, the moving object will stay one more unit of time in (x, y), and then will fly based on its initial route. The rerouting algorithm uses the same procedure as ALG1 to find a route. It first traverses the X dimension and then traverses the Y dimension of the mesh. In contrast to ALG1, ALG2 does not check the time conflict of the aircraft in the future, and therefore backtracking is not needed. Because, we will take care of the conflict by rerouting the aircraft the moment a potential conflict is detected. However, ALG2 does not select a stormy sub-track as a part of its route.

Scenarios. Different parameters such as the rerouting algorithm, the time of the storm, the place of the storm, the network traffic volume, the amount of concurrency arisen from flight plans of the aircraft, and the network dimension change the results of experiments. Because the traffic network of the ATC domain has a cascaded architecture, the place of the storm can be typically approximated as the middle of the traffic network. Therefore, we select the middlemost sub-track of the network as the place of the storm. We perform three sets of experiments; (ES1) that is to compare the time and memory consumptions between the compositional and non-compositional approaches, (ES2) that is to depict the variation of the time consumption in a set of experiments for each approach, and (ES3) that is to compare the scalability of the approaches. The scenarios are described in the following. In our experiments, we assume that FD as the traveling time of an aircraft across a sub-track is one. We also assume that the aircraft consumes one unit of fuel per one unit of the traveling time. Furthermore, the fuel of each aircraft is more than the length of the longest path in the traveling network.

(ES1). We consider a 15×15 mesh structure, divided into 9 regions of 5×5, as the traffic networks in (ES1). The fuel of each aircraft is set to 325. We use ALG1 to generate 150 batches of flight plans per each λ in $\{0.5, 0.25, 0.125\}$, where λ is the parameter of the exponential distribution to generate departure times of the aircraft from source airports. By increasing the value of λ, the mean interval time between two departures decreases. As a result, the network traffic

TABLE 3: The number of experiments in which the model in both approaches faces a deadlock (With Deadlock), in which the model does not face a deadlock and its analysis time is less than a threshold (No Deadlock), in which the verification time passes the threshold (TimeOver). The traffic network has a $n \times n$ mesh structure. λ is the parameter of the exponential distribution to generate the departure times of the aircraft.

n	λ	With Deadlock	No Deadlock or TimeOver	TimeOver
15	0.5	27	120	3
15	0.25	37	110	3
15	0.125	56	94	0

volume and subsequently the concurrency contained in the model might increase. We expect that the compositional approach performs better than the non-compositional approach even in a low concurrency model. Each generated batch contains flight plans of 2000 aircraft. Per each batch $P_i, 1 \leq i \leq 150$, we generate 4 batches $P_{ij}, 1 \leq j \leq 4$, such that P_{i1} contains the first 500 flight plans of P_i, P_{i2} contains the first 1000 flight plans of P_i, and so on. We use both approaches to analyze each batch P_{ij} per each time of the storm in $\{100, 200, 400, 600, 800\}$. Obviously, whenever the storm occurs late, the most of the moving objects have arrived at their destinations. We remove the batch P_i from the experiments of both approaches if for a batch P_{ij} and a time of the storm, the model in one of the approaches is not deadlock-free, or its verification time passes the threshold (the results of experiments in which the models are not deadlock-free are investigated in (ES2)). Table. 3 shows the number of experiments in which both approaches do not face a deadlock and both approaches analyze the model in less than a threshold (No Deadlock or TimeOver), the number of experiments in which both approaches face a deadlock (With Deadlock), and the number of experiments in which the verification time in both approaches passes a threshold (TimeOver). The threshold of the analysis time is set to an hour. In our experiments, per each j, we calculate the averages of the analysis time and the number of states of the batches P_{ij}.

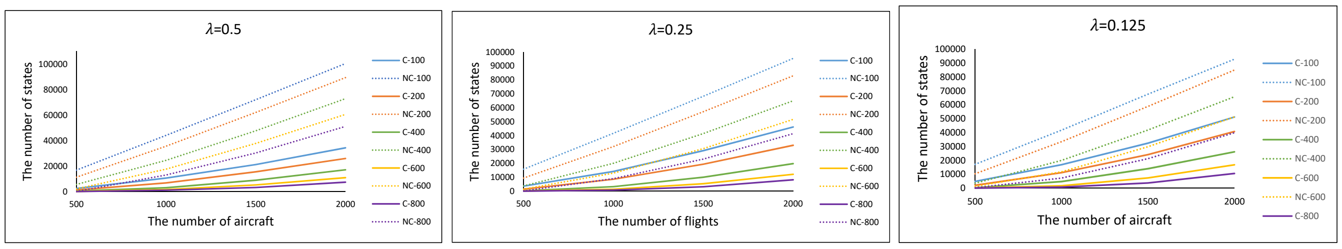
(ES2). The traffic network in (ES2) has the same configuration as the traffic network in (ES1). In (ES2), we use the batches of flight plans generated in (ES1) for $\lambda = 0.5$. The reason for considering $\lambda = 0.5$ is that the network might have the highest traffic volume for $\lambda = 0.5$ compared to $\lambda \in \{0.25, 0.125\}$. We use both approaches to analyze each batch P_i, containing the flight plans of 2000 aircraft. Since the possibility of propagating the change increases when the storm happens early, we suppose that the storm happens at time 100. As shown in Table. 3, the model in 120 experiments is deadlock-free and is analyzed in less than the predefined threshold. Compared to (ES1) that calculates the average of the analysis time for this set of experiments, (ES2) illustrates the variation of the analysis time in this set for each approach. Furthermore, (ES2) depicts the variation of the time consumption to detect a deadlock in 27 experiments that are not deadlock-free.

(ES3). As the aim in (ES3) is to compare the scalability of the approaches, we consider a larger traffic network that is a 18×18 mesh structure with 9 regions of 6×6 in our experiments. The fuel of each aircraft is set to 425. We assume that the change happens at time 100. We use ALG1 to generate a batch P of 5500 flight planes with $\lambda = 0.5$. In (ES3), we start with the first 100 flight plans of P, and gradually increase the number of flight plans to compare the scalability of two approaches. The scalability of the approaches is measured by the number of the aircraft. To this end, We define a threshold for the verification time and set this threshold to 45 minutes. The approach that can analyze a model with more number of the aircraft in less than the defined threshold is more scalable.

6.4 Comparison of the Magnifier approach and the Non-compositional Approach

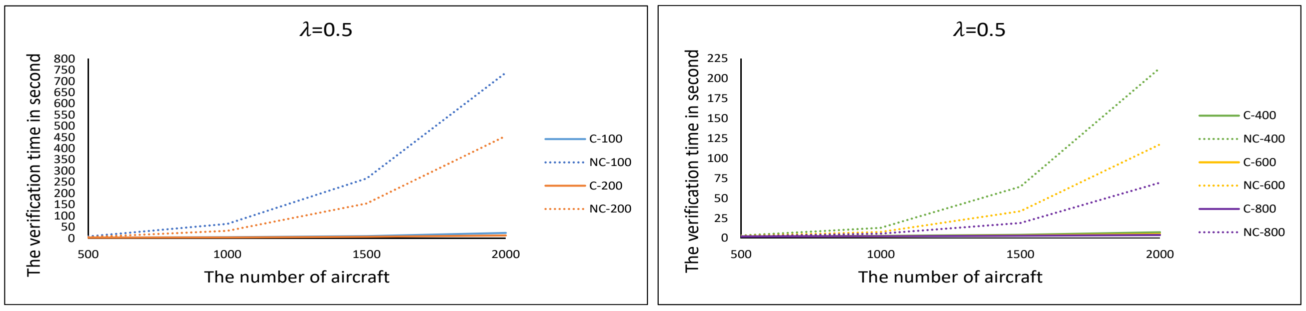
We run our experiments on an ubuntu 18.04 LTS amd64 machine with 67G memory and Intel (R) Xeon (R) CPU E5-2690 v2 @ 3.00GHZ. A part of our experimental results are shown in Figures 6, 7, 8 and 9. In these figures, "C" and "NC" refer to the compositional and non-compositional approaches, respectively. The legend entry $C - i, i \in \{100, 200, 400, 600, 800\}$ depicts the experimental results of the compositional approach for the time i at which the storm happens. The legend entry $NC - i$ depicts the results for the non-compositional approach. As shown in Fig. 6 and Fig. 7, using the compositional approach results in decreasing the verification time and the number of states. As expected, by increasing the number of aircraft, the number of states and accordingly, the verification time increase. The same results are valid for the smaller value of the time at which the storm occurs, since fewer aircraft have arrived at their destinations when the storm happens. By increasing the time at which the storm occurs, the differences between the results of the compositional and non-compositional approaches decrease. It is because most of the aircraft have arrived at their destinations when the storm happens late. To have a better representation of the verification time difference between the compositional and non-compositional approaches, we depict the results of the verification time for $\lambda = 0.5$ in two diagrams with two different time scales, shown in Fig. 7. As can be seen, the compositional approach is able to verify a model with 2000 aircraft in a few seconds for the smallest value of the time at which the storm happens. By increasing the time interval between two departures from a source airport, the number of aircraft entering into the traffic network after the storm happens increases. Therefore, as shown in Fig. 6, the number of states in the compositional approach increases whenever the value of λ decreases.

The results of our experiments in (ES2) are shown in Fig. 8. The variation of the verification time in a set of experiments with no deadlock when the compositional approach is used is shown in Fig. 8(a). The results of the same set of experiments for the case in which the non-compositional approach is used are depicted in Fig. 8(b). We also depict the variation of the time needed to detect a deadlock in a set of experiments using the compositional and non-compositional approaches in Fig. 8(c). As shown in Fig. 8(a), excluding the outliers, the model in our experiments is



(a) (b) (c)

Fig. 6: The number of states in (ES1) for each value of λ in $\{0.5, 0.25, 0.125\}$, where λ is the parameter of the exponential distribution to generate the departure times of the aircraft. The notations C and NC refer to the compositional and non-compositional approaches, respectively. The time at which the storm happens varies in the set $\{100, 200, 400, 600, 800\}$. As an instance, $C - 100$ depicts the results of the compsitional approach when a storm occurs at time 100.



(a) (b)

Fig. 7: The verification time in (ES1) for $\lambda = 0.5$. The left side depicts the verification time in the compositional (C) and non-compositional (NC) approaches when a storm occurs at a time in $\{100, 200\}$. The right side depicts the verification time of each approach when a storm occurs at a time in $\{400, 600, 800\}$. The right and the left side figures show the verification time with different scales.

analyzed in less than 22 seconds using the compositional approach, while this time is around 2190 seconds in the non-compositional approach. Also, in our experiments, the average time for detecting a deadlock in the compositional approach is around 11 minutes, while this value in non-compositional approach is around 20 minutes.

We can define the latency of an adaptation policy by defining a threshold over the analysis time. The latency in our approach is defined as the time needed to adapt the model and to check it for correctness properties. We suppose that a human is involved in adapting the system if the threshold is passed. As an instance, consider that the latency of the adaptation policy in our approach is 3 minutes. From 150 experiments, only 21 experiments (the verification time of 3 experiments pass the threshold, the outlier experiment in Fig. 8(a), and 17 experiments of 27 experiments that face a deadlock) need human intervention in the compositional approach. This is while all experiments in the non-compositional approach need more than 3 minutes analysis time.

The results of our experiment in (ES3) are shown in Fig. 9. To compare the scalabililty of both approaches, we run both approaches for the same scenario. Furthermore, we define a threshold for the verification time and set this threshold to 45 minutes. As can be seen, the non-compositional approach is not scaled for more than 2800 aircraft. The results of the compositional approach in Fig. 9 have fluctuations appeared between 4600 to 5000 aircraft. By adding new aircraft to the traffic network, some areas are congested, and consequently the concurrency of the model increases. This event results in some fluctuations and the fast growth of the "C" plot between 4600 to 5000 aircraft. Except for this range, this plot has a normal growth, since by adding the new aircraft, the behaviors of the congested areas has not sensibly changed.

7 Related Work

In this section, we concentrate on four classes of most related studies. The first class is concerned with the theory of interfaces. The second class is about modeling and verifying traffic control systems. The third class describes the most closely related work that use compositional methods for the verification purpose, and the fourth class is about formal analysis of self-adaptive systems at runtime.

Interface Theory. The theory of interfaces is a widely studied topic. This theory describes the main features that each component-based design should obey, such as refinement, structural composition, and conjunction. The same as ours, the focus of [22] is on the structural composition. The work of [22] presents a theory of timed interfaces to explain the timing constraints on inputs and outputs of the components. A timed interface is encoded as a two-player

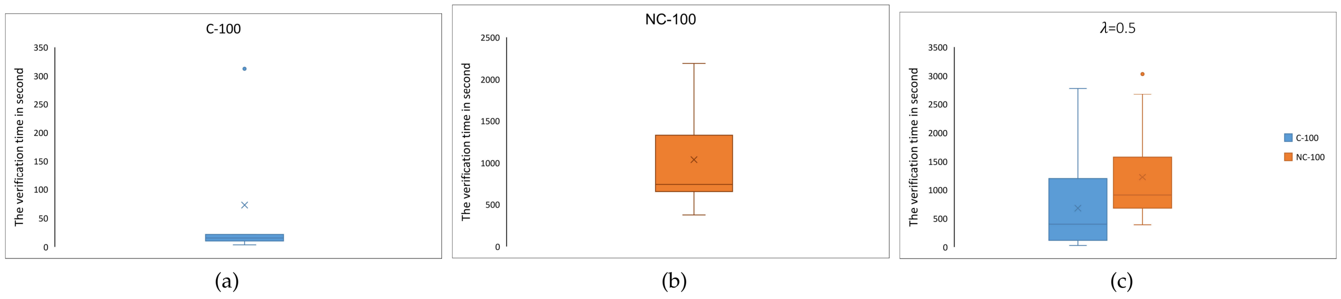


Fig. 8: The verification time in (ES2) for $\lambda = 0.5$. The storm occurs at time 100. The variations of the time needed to verify the experiments with no deadlock using the compositional (C) and non-compositional approaches (NC) are depicted in parts (a) and (b), respectively. The variations of the time needed to detect a deadlock using both approaches are depicted in part (c).

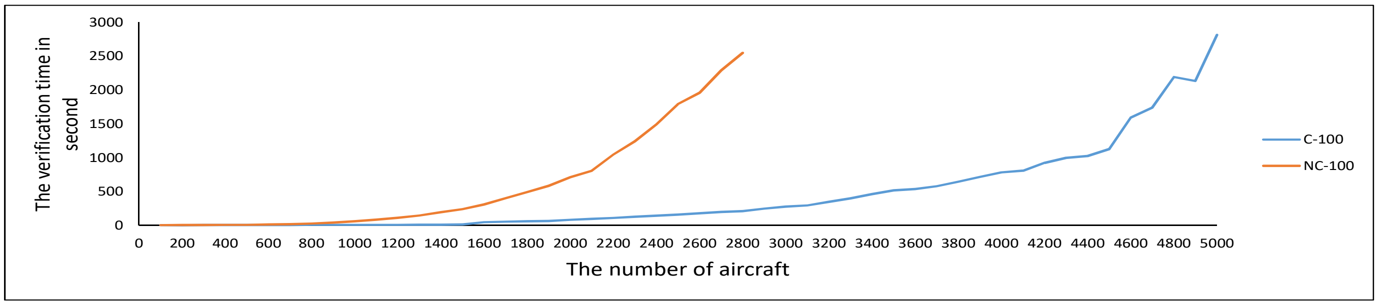


Fig. 9: The scalability of compositional (C) and non-compositional (NC) approaches in (ES3). Both approaches are run for the same scenario with $\lambda = 0.5$. The storm occurs at time 100. The scalability is measured in the number of aircraft, while the verification time is set to a threshold. The non-compositional approach is not able to verify a model with more than 2800 aircraft in a time less than the defined threshold.

timed game in which the environment as the input player provides inputs for the component and the component as the output player creates outputs. This work proposes an optimistic approach of composition that is two components can work together if there is a helpful environment to make them work together. In other words, two components are compatible if the environment has a winning strategy to avoid immediate and time error states in parallel product of two components. An immediate error state is reachable if a component sends an output that is not acceptable by the other component. A component blocks the progress of the time in a time error state. Similar to the approach of [22], [24] proposes an optimistic approach for the structural composition of two interfaces specified by Timed Input Output Transition Systems (TIOTSs). In contrast to [22], [24] assumes that the system is input-enabled. The input-enabled assumption is also considered when interfaces are specified by TIOAs [25]. The approach of composition in [25] is pessimistic. It means that two components should work together in all environments. Optimistic treatment of the composition is also considered in [26], where an interface theory for Modal Input Output Automata is proposed. The approach of composition in [27] is also pessimistic. In [27], two components are compatible if they do not reach an error state in their parallel product. The interfaces in [27] are specified by Modal Input Output transition systems in which the timing constraints are not specified.

Compared to the related work, we follow a pessimistic approach that is two components can work together if parallel product of their TIOAs does not reach a deadlock state. Also the same as [22], we are able to express the input assumptions and there is no need to the input-enabled assumption. Since in our approach the components do not block the progress of the time, we do not need to define time error states.

Modeling and Verifying Traffic Control Systems (TCSs). TCSs such as ATC and train control systems, due to the tight interconnection of the physical plant and the controller software, are mostly categorized as hybrid systems. There is a vast literature on verifying dynamic models of TCSs to detect the future conflicts among the moving objects [28], [29], [30], to resolve the potential conflicts through the trajectory planning [31], [32], and to evaluate the correctness of the communication protocols among different entities of the system [33], [34], [35]. These approaches use the *Lagrangian* models in which the moving objects, e.g. aircraft or trains, along with their operational details are the concern of modeling [36], [37]. As an instance of this kind of modeling, the model of a train control system is created by composing a set of hybrid automata, where each automaton is the model of a train. Modeling the dynamic behaviors of each moving object in these approaches needs a set of differential equations, which due to the large number of the moving objects, makes the analysis of TCSs difficult [36]. Furthermore, this approach of modeling is only necessary when we need to have a microscopic view of the traffic for our analysis purposes.

In contrast to the Lagrangian-based approaches, our approach is based on *Eulerian* models in which the regions of the traveling space, e.g. the sub-tracks in track-based systems, are the concern of modeling [36], [37]. Although this kind of modeling may lose some operational details of the moving objects, it is more appropriate for modeling rerouting/rescheduling of the moving objects [37]. In other words, the adaptation of the system in a macroscopic view is concerned with aggregate behaviors of the moving objects, and affects the whole traffic network by rerouting/rescheduling the moving objects [37]. Therefore, by modeling each subtrack as an actor, we develop a one-dimensional model of the traveling space instead of a complex multi-dimensional model of the moving objects. The properties of our interest such as preventing a moving object from running out of the fuel, and the arrival of a moving object at its destination at a pre-specified time are handled by adding a few features to the message corresponding to the moving object. For instance, each message carries information about the remaining fuel of the moving object for the rest of its travel, or carries the designated time for the arrival of the moving object at each sub-track in its route. This approach of modeling not only provides an acceptable fidelity for the problem [1], but also relieves the analysis difficulties. It is notable that a few of the mentioned approaches such as [28] verify the system at runtime. However, the approach of [28] is not compositional.

Based on the related work, there is an increased interest towards the scheduling and path planning of moving objects in TCSs. A scheduling problem [38], [39] is the problem of efficiently assigning resources to a set of tasks such that some constraints are met, e.g. two tasks do not simultaneously use the same resource. Although the scheduling and path planning are not concerns of this paper, we briefly study the modeling approach of several work by putting the hybrid and dynamic modeling of TCSs aside.

In [38], the same as a sub-track, a resource is in the idle state or in the use state. The resource maintains its use state until a clock reaches a usage time. In [38], [39], Priced Timed Automata (PTA) is used for the scheduling and planning problem. [39] uses PTA for the aircraft landing problem where a landing time and a runway should be assigned to each aircraft. A minimum delay between two aircraft landing on the same runway should be preserved.

In [40], a grid-shaped workspace with static obstacles is shared between a set of robots, where tasks of the system to be performed by the robots are created dynamically. The aim of the paper is to compute an optimal collision-free motion plan for a robot whenever a new task is created. To program these applications, the authors use the P programming language. The robots and also the plan generator are processes of the P language. The concern of this paper is not reducing the model checking time, but decreasing the motion generation time. A multi-robot system is modeled by a network of timed automata in [41]. The approach of [41] obtains all possible trajectories that move robots on a grid-shaped workspace from their initial locations to their destination locations. To this end, each obstacle, each robot, and each controller associated with a robot is modeled by a separate timed automaton. The trajectories are checked against Computational Tree Logic properties in UPPAAL. In [42], a timed automaton for a robot and its environment is created, where each edge shows the movement of the robot and locations are partitions of the space. A property of interest specified in Metric Interval Temporal Logic is transformed into another automaton. The product of the robot automaton and the property automaton is given to UPPAAL. Any execution that starts from an initial state and reaches a final accepting state is an accepting trajectory.

Compared to the related work, we model sub-tracks instead of moving objects. This approach seems to be efficient and scalable when an enormous number of moving objects pass through a fixed workspace. Furthermore, we assume that plans are given or calculated using a planning algorithm, and obstacles are dynamically generated at runtime. However, obtaining optimal plans regarding the harsh timing constraints is a difficult task.

Compositional Methods. In [43], an Assume-Guarantee based approach for verifying self-adaptive systems at design time is proposed. In [43], the changed component is adapted. Then, a backward reasoning starts and regenerates a new assumption for the adapted component. If the new assumption is weaker than the previous assumption of the component, the adaptation is correct. Otherwise, the reasoning continues on the context of the changed component. If it reaches a null assumption on the context of the system, the adaptation is incorrect. The paper focuses on safety properties of the system, and does not consider the change propagation. The work in [44] defines a refinement relation and a weakening operator to check the satisfaction of a property over a real-time system. Each property is divided into a set of subspecifications for which an assumption and a guarantee are defined. The subspecifications, assumptions, and guarantees are defined by TIOAs. The assumption and guarantee are combined into a contract using the weakening operator. The property is satisfied if subspecifications refine their corresponding contracts and vise versa. This approach is not proposed for verifying self-adaptive systems at runtime and consequently does not consider the change and its effects on the system.

Magnifying-Lens Abstraction (MLA), presented in [45], copes with the state space explosion in obtaining the maximal probabilities over a Markov Decision Process (MDP). It partitions the state space into regions, and calculates the upper and lower bounds for the maximal reachability or safety properties on the regions. It magnifies on a region at a time and obtains the values of mentioned parameters by calculating their values for each concrete state. Unlike MLA, the bounds of sub-properties in our approach are given through the interface components. Furthermore, the change propagation is not a concern in [45]. The mechatronic UML (mUML) approach, proposed in [46], uses the refined UML model component and refined state charts to formally define components of a system, their interactions, and their timing and hybrid behaviors. Besides separately checking the safety of each component, mUML checks whether interfaces of components are well-defined and components refine their interfaces. In contrast to our approach, this approach is able to model hybrid behaviors, but change propagation is not considered in [46]. Furthermore, as explicitly mentioned in the paper, verification at runtime is not a concern in [46].

The ACPS language to design and verify self-adaptive CPSs is proposed in [47]. The components of the system are categorized in different groups such that each group affects on satisfaction of one requirement. An adaptation is encoded for each group such that the requirement is preserved. Finally, the ACPS definition of each group is translated to a verification tool and is separately verified. This work assumes that grouping the relevant components to a requirement is possible. In contrast to [47], instead of considering a fixed number of components per each requirement, we increase the verification domain whenever it is needed. Furthermore, [47] does not consider the change propagation phenomenon.

Formal Analysis of Self-adaptive Systems at Runtime. Incremental runtime verification of MDPs, described in the PRISM language, is proposed in [48], where runtime changes are limited to vary parameters of the PRISM model. An MDP is constructed incrementally by inferring a set of states needed to be rebuilt. The constructed MDP is then verified using an incremental verification technique. Runtime verification of parametric Discrete Time Markov Chains (DTMCs) is accomplished in [49]. In this method, probabilities of transitions are given as variables. Then, the model is analyzed and a set of symbolic expressions is reported as the result. By substituting real values of the variables at runtime, verification is reduced to calculating the values of the symbolic expressions.

In [50], a self-adaptive software is designed as a dynamic software product line (DSPL). Then, an instance of DSPL is chosen at runtime considering the environmental changes. This approach uses parametric DTMCs to model common behaviors of the products and each variation point separately. Therefore, there is no need to verify each configuration separately. RINGA, introduced in [51], uses Finite State Machines (FSM) to develop a design-time model of a system, and abstracts the model for using at runtime. Each state of the model implements a module, while a transition triggers an adaptation. Each transition is assigned an equation that is parameterized by environmental variables. The value of the equation is calculated at runtime. Lotus@runtime [52] uses Probabilistic Labeled Transition Systems (PLTS) to develop a model@runtime. It monitors execution traces of the system and updates the probabilities in PLTS. The desirable properties in [52] are explained through a source state, a target state, a condition to be satisfied, and the probability of satisfying the condition.

In comparison to [48], [49], [50], [51], [52] which use state-based models, an actor model is in a higher level of abstraction. Our actor-based approach besides decreasing the semantic gap between the model@runtime and applications, facilitates the modular analysis of the system.

The failure propagation is studied in [53] that checks whether the structural adaptation of the system is fast enough to prevent a hazard. After an adaptation, it is checked whether the remaining failures in the system lead to a hazard. Our approach, besides detecting a hazard, assures the satisfaction of the timing properties of the system. Based on the circumstances existing at the time the change occurs, different thresholds over the analysis time can be imposed. It is assumed that humans are involved if the adaptation cannot be handled during the expected time. The latency-aware adaptation is studied in [54], where a probabilistic model checker proactively selects an adaptation strategy to maximize the utility of the system. Unlike [54], our focus is on effectively verifying the system behavior.

The work of [55] investigates which state of the system is a safe state to update the implementation of the system whenever an environment assumption is changed. Furthermore, based on the old controller, a new controller is automatically synthesized for the software system. The approach of [55] is applied on a RailCab system where an accident should be avoided before the RailCabs enter into a crossing. An infrastructure to deploy and execute new controllers on embedded devices is proposed in [56]. [55] does not verify the system after adapting it to a change.

8 Discussion and Future Work

We proposed Magnifier, a compositional approach that iteratively detects the propagation of a change and incrementally involves the components affected by a change into the analysis. An adaptation policy may contain the change and prevent the change to be propagated. In the worse case, the change propagates to the whole system and Magnifier needs to compose all components of the model. We compared the compositional approach of Magnifier and the non-compositional approach in Section 6. The comparisons between our model, CoodAA, and other similar models on self-adaptive systems are presented in [1], [3]. In Section 7, we included a comparison of Magnifier with other analysis approaches for TTCS, and other compositional methods.

Here we include an observation regarding a comparison between the non-compositional approach and the worst case for Magnifier when the change propagates all the way to include the whole system. We argue that even in the worst case Magnifier performs better than the non-compositional approach. Our experiments testify this observation. Looking more carefully into this comparison and building a formal proof is a part of our future work. This observation can be justified as follows. Suppose that a change happens at time t. The non-compositional approach involves all actors of the model that have a message at time t into the analysis, and starts to generate the state space. In contrast, Magnifier focuses on the component affected by the change, and starts to generate the state space by involving those actors of the component having a message at time t. Let the branching factor for a state be the number of outgoing transitions of the state or the number of actors that can be triggered at the state. At the beginning, Magnifier has a lower branching factor on all states of the state space compared to the non-compositional approach. At some point in future, e.g. at time t', when all components are affected by the change, both approaches involve the same number of actors into the analysis, and both approaches generates the same number of states and transitions. However, between t and t', the graph of the state space in Magnifier is smaller than the graph of the state space in the non-compositional approach. Therefore, even in the worst case, Magnifier performs better than the non-compositional approach in terms of the verification time and the memory consumption.

Our Ptolemy II implementation of Magnifier is specialised for ATC. In [37], Lee and Sirjani show how CoodAA can capture TTCS applications in general. Here we consider a constant number of four for the ports of all actors, and the topology formed by connecting the ports through channels is a mesh. The extension to a dynamic number of ports and further than to dynamic bindings, seem like natural future work. The general idea of Magnifier is not limited to TTCSs. It can be generalised for any control system with a modular design. We need to extend our model to include more general actors with different behaviors, and also different number of ports, and different bindings among the ports (channels that form the topology). To investigate the details of such extension is another future direction. The possibility of analyzing actors in a compositional way is a consequence of their isolation discussed in [57] by Sirjani, Khamespanah and Ghassemi. Hence, we believe that CoodAA and Magnifier can be further extended and used in different areas and applications based on the foundations provided in this paper.

Acknowledgments

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. The work of the second author is supported in part by KKS SACSys Synergy project (Safe and Secure Adaptive Collaborative Systems), KKS DPAC Project (Dependable Platforms for Autonomous Systems and Control), and SSF Serendipity project at Malardalen University, and MACMa Project (Modeling and Analyzing Event-based Autonomous Systems) at Software Center, Sweden. The authors would like to thank Professor Edward A. Lee for his valuable suggestions.

References

[1] M. Bagheri, M. Sirjani, E. Khamespanah, N. Khakpour, I. Akkaya, A. Movaghar, and E. A. Lee, "Coordinated actor model of self-adaptive track-based traffic control systems," *Journal of Systems and Software*, vol. 143, pp. 116 – 139, 2018.

[2] 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, and N. M. Villegas, *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*. Cham: Springer International Publishing, 2014, pp. 101–136.

[3] M. Bagheri, I. Akkaya, E. Khamespanah, N. Khakpour, M. Sirjani, A. Movaghar, and E. A. Lee, "Coordinated actors for reliable self-adaptive systems," in *Formal Aspects of Component Software: FACS 2016*, O. Kouchnarenko and R. Khosravi, Eds., 2017, pp. 241–259.

[4] C. Hewitt, "Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot," MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, Tech. Rep., 1972.

[5] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

[7] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, "Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems," in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*. IEEE, 2003, pp. 166–177.

[8] E. M. Clarke, D. E. Long, and K. L. McMillan, "Compositional model checking," in *Logic in Computer Science. LICS'89, Proceedings., Fourth Annual Symposium on*. IEEE, 1989, pp. 353–362.

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

[10] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, no. Part B, pp. 184 – 206, 2015, 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian.

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

[12] M. Bagheri, E. Khamespanah, M. Sirjani, A. Movaghar, and E. A. Lee, "Runtime compositional analysis of track-based traffic control systems," *SIGBED Rev.*, vol. 14, no. 3, pp. 38–39, Nov. 2017.

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

[14] A. Zbrzezny and A. Półrola, "Sat-based reachability checking for timed automata with discrete data," *Fundamenta Informaticae*, vol. 79, no. 3-4, pp. 579–593, 2007.

[15] S. Tripakis, "Verifying progress in timed systems," in *Formal Methods for Real-Time and Probabilistic Systems*, J.-P. Katoen, Ed. Springer Berlin Heidelberg, 1999, pp. 299–314.

[16] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal 4.0," *Department of computer science, Aalborg university*, 2006.

[17] N. Khakpour, S. Jalili, C. Talcott, M. Sirjani, and M. Mousavi, "Formal modeling of evolving self-adaptive systems," *Science of Computer Programming*, vol. 78, no. 1, pp. 3 – 26, 2012, special Section: Formal Aspects of Component Software (FACS'09).

[18] N. Khakpour, S. Jalili, M. Sirjani, U. Goltz, and B. Abolhasanzadeh, "Hpobsam for modeling and analyzing it ecosystems – through a case study," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2770 – 2784, 2012, self-Adaptive Systems.

[19] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and verification of reactive systems using rebeca," *Fundam. Inform.*, vol. 63, no. 4, pp. 385–410, 2004.

[20] E. Khamespanah, M. Sirjani, M. Viswanathan, and R. Khosravi, "Floating time transition system: More efficient analysis of timed actors," in *FACS*, 2015, pp. 237–255.

[21] E. Khamespanah, M. Sirjani, Z. Sabahi-Kaviani, R. Khosravi, and M. Izadi, "Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system," *Sci. Comput. Program.*, vol. 98, pp. 184–204, 2015.

[22] L. de Alfaro, T. A. Henzinger, and M. Stoelinga, "Timed interfaces," in *Embedded Software*, A. Sangiovanni-Vincentelli and J. Sifakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 108–122.

[23] S. D. Chawade, M. A. Gaikwad, and R. M. Patrikar, "Review of xy routing algorithm for network-on-chip architecture," *International Journal of Computer Applications*, vol. 43, no. 21, pp. 975–8887, 2012.

[24] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed i/o automata: a complete specification theory for real-time systems," in *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*. ACM, 2010, pp. 91–100.

[25] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, "Timed i/o automata: a mathematical framework for modeling and analyzing real-time systems," in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, Dec 2003, pp. 166–177.

[26] F. Bujtor, "Modal interface automata: A theory for heterogeneous specification of parallel systems," Ph.D. dissertation, University of Augsburg, Germany, 2018.

[27] S. S. Bauer, R. Hennicker, and S. Janisch, "Interface theories for (a)synchronously communicating modal i/o-transition systems," in *Proceedings Foundations for Interface Technologies, FIT 2010, Paris, France, 30th August 2010.*, 2010, pp. 1–8.

[28] L. Bu, Q. Wang, X. Chen, L. Wang, T. Zhang, J. Zhao, and X. Li, "Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior," *SIGBED Rev.*, vol. 8, no. 2, pp. 7–10, Jun. 2011.

[29] H. A. P. Blom, J. Krystul, and G. J. Bakker, "A particle system for safety verification of free flight in air traffic," in *Proceedings of the 45th IEEE Conference on Decision and Control*, 2006, pp. 1574–1579.

[30] R. Cheng, J. Zhou, D. Chen, and Y. Song, "Model-based verification method for solving the parameter uncertainty in the train control system," *Reliability Engineering and System Safety*, vol. 145, pp. 169 – 182, 2016.

[31] S. M. Loos, D. Renshaw, and A. Platzer, "Formal verification of distributed aircraft controllers," in *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '13. ACM, 2013, pp. 125–130.

[32] K. Margellos and J. Lygeros, "Toward 4-d trajectory management in air traffic control: A study based on monte carlo simulation and reachability analysis," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 5, pp. 1820–1833, 2013.

[33] W. Damm, A. Mikschl, J. Oehlerking, E.-R. Olderog, J. Pang, A. Platzer, M. Segelken, and B. Wirtz, *Automating Verification of Cooperation, Control, and Design in Traffic Applications*. Springer Berlin Heidelberg, 2007, pp. 115–169.

[34] Y. Zhao and K. Y. Rozier, "Formal specification and verification of a coordination protocol for an automated air traffic control system," *Science of Computer Programming*, vol. 96, pp. 337 – 353, 2014, special Issue on Automated Verification of Critical Systems (AVoCS 2012).

[35] "Performance analysis and verification of safety communication protocol in train control system," *Computer Standards and Interfaces*, vol. 33, no. 5, pp. 505 – 518, 2011.

[36] P. K. Menon, G. D. Sweriduk, and K. D. Bilimoria, "New approach for modeling, analysis, and control of air traffic flow," *Journal of guidance, control, and dynamics*, vol. 27, no. 5, pp. 737–744, 2004.

[37] E. A. Lee and M. Sirjani, "What good are models?" in *Formal Aspects of Component Software*, K. Bae and P. C. Ölveczky, Eds. Cham: Springer International Publishing, 2018, pp. 3–31.

[38] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Optimal scheduling using priced timed automata," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 4, pp. 34–40, Mar. 2005.

[39] J. I. Rasmussen, K. G. Larsen, and K. Subramani, "On using priced timed automata to achieve optimal scheduling," *Formal Methods in System Design*, vol. 29, no. 1, pp. 97–114, Jul 2006.

[40] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia, "Drona: A framework for safe distributed mobile robotics," in *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*, April 2017, pp. 239–248.

[41] M. M. Quottrup, T. Bak, and R. I. Zamanabadi, "Multi-robot planning : a timed automata approach," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, vol. 5, April 2004, pp. 4417–4422 Vol.5.

[42] Y. Zhou, D. Maity, and J. S. Baras, "Timed automata approach for motion planning using metric interval temporal logic," in *2016 European Control Conference (ECC)*, June 2016, pp. 690–695.

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

[44] A. David, K. G. Larsen, A. Legay, M. H. Møller, U. Nyman, A. P. Ravn, A. Skou, and A. Wksowski, "Compositional verification of real-time systems using ecdar," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 6, pp. 703–720, Nov 2012.

[45] L. de Alfaro and P. Roy, "Magnifying-lens abstraction for markov decision processes," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., 2007, pp. 325–338.

[46] H. Giese and W. Schäfer, *Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 152–186.

[47] A. Borda, L. Pasquale, V. Koutavas, and B. Nuseibeh, "Compositional verification of self-adaptive cyber-physical systems," in *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2018, pp. 1–11.

[48] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "Incremental runtime verification of probabilistic systems," in *Runtime Verification*, ser. LNCS, S. Qadeer and S. Tasiran, Eds., 2013, vol. 7687, pp. 314–319.

[49] A. Filieri and G. Tamburrelli, "Probabilistic verification at runtime for self-adaptive systems," in *Assurances for Self-Adaptive Systems*, ser. LNCS, J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, Eds., 2013, vol. 7740, pp. 30–59.

[50] C. Ghezzi and A. Molzam Sharifloo, "Dealing with non-functional requirements for adaptive systems via dynamic software product-lines," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS, R. de Lemos, H. Giese, H. Müller, and M. Shaw, Eds., 2013, vol. 7475, pp. 191–213.

[51] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D.-K. Baik, "Ringa: Design and verification of finite state machine for self-adaptive software at runtime," *Information and Software Technology*, vol. 93, no. Supplement C, pp. 200 – 222, 2018.

[52] D. M. Barbosa, R. G. D. M. Lima, P. H. M. Maia, and E. Costa, "Lotus@runtime: A tool for runtime monitoring and verification of self-adaptive systems," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2017, pp. 24–30.

[53] C. Priesterjahn, D. Steenken, and M. Tichy, "Timed hazard analysis of self-healing systems," in *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*, J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, Eds., 2013, pp. 112–151.

[54] G. A. Moreno, J. Cámara, D. Garlan, and 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, pp. 1–12.

[55] C. Ghezzi, J. Greenyer, and V. P. L. Manna, "Synthesizing dynamically updating controllers from changes in scenario-based specifications," in *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, June 2012, pp. 145–154.

[56] V. P. La Manna, J. Greenyer, D. Clun, and C. Ghezzi, "Towards executing dynamically updating finite-state controllers on a robot system," in *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*, ser. MiSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 42–47.

[57] M. Sirjani, E. Khamespanah, and F. Ghassemi, "Reactive actors: Isolation for efficient analysis of distributed systems," in *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2019, pp. 1–10.

Maryam Bagheri received the M.S. degree in computer engineering from Sharif University of Technology, Tehran, Iran, in 2013. She is currently working toward the Ph.D. degree in computer engineering at the Department of Computer Engineering at Sharif University of Technology.

Marjan Sirjani is a Professor and chair of Software Engineering at Mälardalen University, and the leader of Cyber-Physical Systems Analysis research group. She is also a part-time Professor at School of Computer Science at Reykjavik University. Her main research interest is applying formal methods in Software Engineering. She works on modeling and verification of concurrent, distributed, and self-adaptive systems. Marjan and her research group are pioneers in building model checking tools for actor models.

Ehsan Khamespanah is a graduate student from a double-degree program in the ECE Department at Tehran University and the department of computer science at Reykjavik University. He is a Postdoctoral Researcher in Software Architecture and Formal Methods lab at Tehran University. His research interests include formal methods, software testing, cyber-physical systems, and software architecture. Ehsan has a BE in computer engineering form Tehran University.

Christel Baier is a full professor and head of the chair for Algebraic and Logic Foundations of Computer Science at the Faculty of Computer Science of the Technische Universität Dresden since 2006. From the University of Mannheim she received her Diploma in Mathematics in 1990, her Ph.D. in Computer Science in 1994, and her Habilitation in 1999. She was an associate professor for Theoretical Computer Science at the University of Bonn from 1999 till 2006.

Ali Movaghar received the B.S. degree in electrical engineering from the University of Tehran, in 1977, and the M.S. and Ph.D. degrees in computer, information, and control engineering from the University of Michigan, Ann Arbor, in 1979 and 1985, respectively. He is a professor in the CE Department, Sharif University of Technology, in Tehran, Iran. His research interests include performance/dependability modeling and formal verification of distributed real-time systems. He is a senior member of the IEEE and the ACM.