Runtime Compositional Verification of Self-adaptive Track-based Traffic Control Systems

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

Abstract—In this paper, we propose a compositional approach for verifying autonomous Track-based Traffic Control Systems at runtime. This approach traces a sequence of *changes* propagated through the system and uses compositional verification to verify the changed/adapted components. A component of the system is modeled by a coordinated actor model and interacts with several components, called its environment components. We define the operational semantics of a coordinated actor model based on Timed Input Output Transition System (TIOTS). We call two (or more) TIOTSs composable if they do not reach a deadlock in their parallel composition. By detecting a change in a component, the component is adapted. If TIOTSs of the adapted component and its environment components are composable, the change does not propagate to the environment components and properties of the system are preserved. Otherwise, the change is propagated. In this case, all components affected by the change are adapted and are composed to form a new component. It is then checked whether TIOTSs of the new component and its environment components are composable. This procedure continues until the change does not propagate. To reduce the state space, for checking the composability we use a reduced version of the TIOTSs of the environment components. We implement our approach in the Ptolemy II framework. The results of our experiments indicate that the proposed approach improves the model checking time and the memory consumption.

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

1 Introduction

AUTONOMOUS response to context changes is a distinguishing feature of self-adaptive systems, where a system is able to adjust its structure and behavior in response to changes in its environment and the system itself. Following a change, a sequence of changes may happen. In other words, to satisfy expected properties of a system, the system is adapted, while the adaptation may result in further changes in the system. For each change and its consequent adaptation, verifying the safety and quality properties of the self-adaptive system is necessary. Due to uncertainties in the context of a self-adaptive system, its reliability should be checked during its execution. Towards this aim, verification at runtime is recommended, where an abstract model of the system and its environment is designed as the model@runtime, is updated, and is verified during the system execution. The analysis results are used to develop an adaptation plan and the plan is issued to the system.

Verification at runtime due to its nature to be performed at runtime has strict time and memory constraints. Although there are approaches for verifying self-adaptive systems at runtime [1], [2], [3], [4], [5], [6], a few of them cope with these constraints and none of them study the *change propagation* phenomenon, where following a change in the context of a system, a sequence of changes may happen in the system. To deal with the time and memory constraints in checking a system at runtime, we propose an approach based on compositional verification [7], [8], where satisfaction of a global property over the whole system is deduced from satisfaction of local properties over components of the system. Our approach by detecting a propagation of the change through the system, dynamically extends its verifying domain to check the changed/adapted components.

Using the approach of this paper, the model of a system is decomposed into a set of components, and the expected behavior of the system is demonstrated by parallel composition of the components. The behaviors of these components is presented by Timed Input Output Transition Systems (TIOTSs) [9]. This way, the problem of parallel composition of the components is reduced to the problem of parallel composition of their corresponding TIOTSs. We call two (or more) TIOTSs composable if they do not reach a deadlock in their parallel composition. Accordingly, we call two (or more) components composable, if their corresponding TIOTSs are composable.

Upon encountering a change, the affected component is adapted to the change. Each component of the model interacts with a set of components called its environment components. If the adapted component and its environment components are composable, it means that the adaptation contains the change and prevents the possible effects of the change to be propagated to other components. In contrast, the adaption may result in some disturbances in the environment components. In this case, the adapted component and its environment components are not composable and the change is propagated. To eliminate the effects of the

M. Bagheri and A. Movaghar are with the Department of Computer Engineering, Sharif University of Technology, Iran. M. Sirjani is with the School of IDE, 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.
Manuscript received XX, 2018; revised XX, XXXX.

change propagation, more adaptations by the environment components are needed. Now, the affected component and its adapted environment components are composed to form a new component. Similarly, if the new component and its environment components are composable, it shows that the set of adaptations successfully contains the change and there is no more change propagation. Otherwise, the change is propagated to more components

Our approach is particularly proposed for autonomous Track-based Traffic Control Systems (TTCSs). TTCSs are large-scale, cyber-physical, safety and time critical systems in which the change propagation phenomenon is clearly visible as described in Section 2. In TTCSs, traffic passes through pre-specified tracks that based on the safe distance between the moving objects are divided into a set of sub-tracks. The sub-tracks are critical sections, accommodating only one moving object in-transit. The controller in a TTCS coordinates the moving objects by safely routing/rerouting/rescheduling them. In [10], we introduced a coordinated actor model to build the model@runtime of a self-adaptive TTCS. In the coordinated actor model of a TTCS, each sub-track is modeled by an actor, the moving objects are considered as messages passed by the actors, and the controller is modeled by a coordinator, explained in Section 3. Since a TTCS is a large-scale system, it is divided into several control areas, and each control area has its own controller. To model large-scale TTCSs, we proposed a hierarchical model for multiple interactive coordinated actor models [11] consisting of a coordinated actor model per each area. The collaboration between the models is achieved through their coordinators and message passing among the actors.

Compositional verification is more effectively applicable to the multiple interactive coordinated actor models, since each component as a coordinated actor model interacts with its environment based on predefined interfaces. Each component should guarantee to be able to receive messages from its environment components and to send messages to its environment components at the pre-specified times. To exploit our approach for verifying self-adaptive TTCSs, we give the operational semantics of a coordinated actor model based on TIOTSs, described in Section 4. We also define the parallel composition of two TIOTSs and explain when their composition reaches a deadlock. To detect the change propagation, we check whether the adapted component and its environment components are composable, described in Section 5. To reduce the state space, TIOTSs of the environment components are reduced, since only the transitions related to sending messages from the environment to the component and vice versa are important.

To illustrate the applicability of our approach, we implement it in Ptolemy II [12]. 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 a Model of Computation (MoC), implemented in a director component. Ptolemy provides a library of deterministic MoCs. In [11], we developed a Ptolemy template to model and analyze self-adaptive TTCSs. In this paper, we develop a director that supports assertion-based verification of TTCSs. Our director generates the state space of the affected component, automatically extends its verifying domain to include several components, and performs the reachability analysis. The results of our experiments for an example in the domain of Air Traffic Control system (ATC) indicate a significant improvement in the time and memory consumptions, described in Section 6.

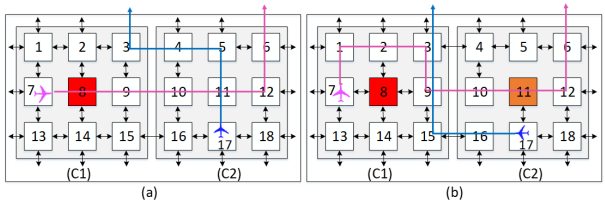

Fig. 1: An ATC example with 18 sub-tracks. The effect of the storm in sub-track 8 is propagated to component c_2. To avoid the collision in sub-track 11, the blue aircraft is rerouted.

In [13], we introduced the notion of a Magnifier for runtime compositional verification of self-adaptive TTCSs as a work in progress. In this paper, we develop the theoretical part of the Magnifier based on TIOTSs, and implement it in the Ptolemy II framework. We describe the related work in Section 7, and we conclude the paper in Section 8.

2 Problem Definition

An ATC is a system equipped with supervision instruments that monitor and control flights along the airspace routes. ATC in the North Atlantic follows a track-based structure that is called an Organized Track System (OTS) [14]. The North Atlantic OTS consists of a set of nearly parallel tracks positioned in light of the prevailing winds to suit the traffic between Europe and North America. Based on the safe distance between two aircraft, the tracks are divided into a set of sub-tracks. Each sub-track is a critical section that accommodates only one aircraft in-transit. ATC uses this structure to guarantee the safety and improve the performance. In the real world, each aircraft has an initial flight plan. An aircraft flight plan consists of its flight route as a sequence of the sub-tracks flown by the aircraft from its source airport to its destination, initial fuel, and time schedule decisions. The aircraft time schedule decisions consist of its departure time from its source airport, assumed arrival time at each sub-track in its route, and assumed arrival time at the destination airport. The initial flight plans are generated prior to takeoff, but dynamic changes in the weather conditions, delay in landing and taxiing, etc., may require some modifications in the aircraft flight plans. In other words, following a change in the airspace, a sequence of changes might happen. For instance, the aircraft flight plans are changed if a storm happens in a part of their flight routes. While changing aircraft flight plans, several safety issues should be considered; i.e. loss of the separation between two aircraft should be avoided, and the remaining fuel should be checked. To avoid conflicts, changing the flight plan of an aircraft may result in changing the flight plans of the other aircraft. These changes can be propagated to the whole system. Besides the safety concerns, performance metrics such as arrival times of the aircraft at their destinations or sub-tracks in their

routes are important. In ATC, the controller is in charge of coordinating the aircraft by routing or rerouting them.

An example of the change propagation in an ATC system is described as follows. Assume Fig. 1(a) and Fig. 1(b) show a part of an ATC example with 18 sub-tracks (two areas). 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 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, the aircraft cannot cross over the sub-tracks affected by the storm and are rerouted. 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). Suppose that the traveling times of the moving objects through each sub-track are the same and are equal to one. The initial flight routes and time schedule decisions of the purple and blue aircraft in Fig. 1(a) are $\{\{7,8,9,10,11,12,6\}\{0,1,2,3,4,5,6\}\}$ and $\{\{17,11,5,4,3\}\{5,6,7,8,9\}\}$, respectively. For instance, the purple aircraft arrives at sub-track 7 at time zero and exits it at time one which is equal to its arrival time at sub-track 8. By the occurrence of a storm in sub-track 8, the route and time schedule of the purple aircraft are changed to $\{\{7,1,2,3,9,10,11,12,6\}\{0,1,2,3,4,5,6,7,8\}\}$, shown in Fig. 1(b). Thus, the purple aircraft arrives at sub-track 11 at time 6. At this time, the blue aircraft has to enter into sub-track 11 based on its initial flight route. To prevent the collision between two aircraft, the controller employs a rerouting algorithm (adaptation policy) and changes the route and time schedule of the blue aircraft to $\{\{17,16,15,9,3\}\{5,6,7,8,9\}\}$. 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 aircraft.

The described track-based structure of ATC is followed in many applications such as the rail traffic control systems, maritime transportation, smart hubs, unmanned vehicles, and centralized robotic systems with the mesh structure. In these systems, to reduce the risk of collision between the moving objects, the traveling space is divided into smaller safe regions, and a centralized controller manages the traffic flow. In [10], we introduced these systems as TTCSs, where the small safe regions are called tracks. Each track is divided into several sub-tracks. The change propagation is a common phenomenon among TTCSs. As a change in the context of a TTCS and its consequent adaptations in the system happen at runtime, several questions arise. For instance, how is the separation between two moving objects guaranteed (a safety property)? Regarding the designed adaptation (selected rerouting algorithm), does the blue aircraft arrive at sub-track 3 at time 9 (qualitative property)? can the controller design an adaptation plan (select a rerouting algorithm among its algorithms) to satisfy the given properties (synthesis)? These questions can be answered by using verification at runtime. However, TTCSs are large scale and verifying given properties at runtime faces state space explosion. The proposed approach in this paper is our first step toward addressing this problem using compositional verification.

3 PRELIMINARIES

In this section, we briefly recall the definitions of Timed Transition System (TTS) [15] and Timed Input Output Transition System (TIOTS) [9], [16]. We also introduce the coordinated actor model and multiple interactive coordinated actor models as the high-level models whose semantics will be defined based on TTS.

3.1 Timed Transition System

A TTS is a 5-tuple $\Pi = (S, s_0, Act, \rightarrow, AP, L)$, where S is the set of states, s_0 is the initial state, Act is the set of actions, $\rightarrow \subseteq S \times (Act \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation, AP is the set of atomic propositions, and L is the labeling function. We use $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \rightarrow$. The transition $s \xrightarrow{a} s'$ is a discrete transition if $a \in Act$. Otherwise, it is called a timed transition. A TTS with the partitioned action set $Act = Act_I \cup Act_O \cup \{\tau\}$ is called a TIOTS, where Act_I and Act_O are the sets of input and output actions, respectively. We use $\Pi = (S, s_0, Act_I, Act_O, \rightarrow, AP, L)$ to denote a TIOTS.

3.2 Coordinated Actor Model

Actors [17] are distributed, autonomous objects that interact by asynchronous message passing. In [10], we introduced the coordinated actor model to realize self-adaptive TTCSs. The coordinated actor model as an extension of the actor model encapsulates a set of actors together with a coordinator. The coordinator uses a scheduler to govern message passing among the actors. In comparison with the actor model, 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 scheduler. The scheduler selects an event from its buffer based on a given policy and delivers the message to its receiver actor. The same as the actor model, the receiver actors pick the delivered messages and process them.

The coordinated actor model is designed based on the MAPE-K feedback loop [18]. Using the MAPE-K feedback loop is a common approach for realizing self-adaptive systems. This control loop consists of the Monitor, Analyze, Plan, and Execute components together with the Knowledge part. The model@runtime is kept in the *Knowledge* part, is updated by the *Monitor* component, and is analyzed by the *Analyze* component. Based on the analysis results, the *Plan* component makes an adaptation plan that is sent to the system through the *Execute* component. In [10], the coordinator is extended by a decision maker. The decision maker encompasses the Analyze and Plan activities of the MAPE-K feedback loop. The actors along with the scheduler construct the model@runtime. The decision maker is able to execute the model@runtime to investigate the future behavior of the system. It then applies its decision to the model@runtime and the system.

The coordinated actor model is aligned with the structure of TTCSs, since 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. The controller (coordinator) is able to reroute/reschedule the moving objects considering the congestion and environmental conditions. It also can be augmented with

several rerouting/rescheduling algorithms, and by predicting the behavior of the system through executing the model@runtime, selects the best algorithm for rerouting/rescheduling purpose [10].

In [11], we developed a Ptolemy template to model and analyze self-adaptive TTCSs based on the coordinated actor model. In our template, each sub-track is modeled by a Ptolemy actor, and the controller is modeled by a Ptolemy director. In this paper, we need to develop a director that not only supports the compositional reasoning, also defines the nondeterministic semantics arisen from the concurrent execution of the actors.

3.3 Multiple Interactive Coordinated Actor Models

Multiple interactive coordinated actor models consist of a coordinated actor model per each subsystem and a top-level coordinator [11]. Each coordinated actor model has its own actor-based model@runtime. The interactions of the coordinators are managed by the top-level coordinator. The message passing among different coordinated actor models is governed by the scheduler of the top-level coordinator. We use the multiple interactive coordinated actor models to build a large-scale TTCS. A TTCS is divided into several control areas. Each area has its own controller and is modeled by a coordinated actor model. In other words, adaptations in a large-scale TTCS cannot be handled by a centralized MAPE-K feedback loop and developing several interactive MAPE-K feedback loops is necessary.

4 Syntax and Semantics of the Coordinated Actor Model

In this section, we provide formal specifications for the syntax and semantics of the (multiple interactive) coordinated actor model(s). The operational semantics of the coordinated actor model is presented in terms of TIOTS. We also define parallel composition of two TIOTSs in the context of our study and illustrate it by an example.

4.1 Abstract Syntax of the Coordinated Actor Model

Prior to proposing the abstract syntax of the coordinated actor model, we present the notations used in the rest of this section. Given a set B, B^* is the set of all finite sequences over elements of B, and $\mathcal{P}(B)$ is the power set of B. A coordinated actor model $CAM = (A, c, CH)$ contains the set A of actors, coordinator c, and set CH of channels. Each connection between two actors is called a channel and includes a queue of messages that are sent by an actor over the channel to another actor. Each channel is an instance of $Channels = ChId \times AId \times Msg^* \times AId$, where $ChId$ is the set of all channel identifiers, AId is the set of all actor identifiers, and Msg is the set of all messages in the model. A channel $(ch, i, msgs, j)$ has the identifier ch and a sequence of messages that are sent from the actor a_i to the actor a_j, where i and j are the identifiers of the connected actors through the channel. Each actor is an instance of $Actors = AId \times MtDs \times MtDs \times \mathcal{P}(Vars) \times \mathcal{P}(ChId) \times \mathcal{P}(ChId)$, where $MtDs$ is the set of all method declarations and $Vars$ is the set of all variable names. An actor $a_i = (i, initialize, fire, vars, Ch_{I_i}, Ch_{O_i})$ has the identifier i, the method $initialize$, the method $fire$, the set of state variables $vars$, and the set of input and output channels Ch_{I_i} and Ch_{O_i}, respectively. The $initialize$ method initializes variables of the actor, and the main computation of the actor is defined in its $fire$ method. An actor can send messages over its output channels or read messages from its input channels during the execution of its $fire$ method. Furthermore, the actor can execute the $selfCall$ statement in its $fire$ and $initialize$ methods to ask the coordinator to fire it again at a given time. By executing the $selfCall$ statement, the actor sends an empty message to itself at a given time. The coordinator is an instance of $C = CId \times \mathcal{P}(Vars) \times MtDs \times MtDs \times \mathcal{P}(MtDs) \times \mathcal{P}(E) \times \mathbb{R}_{\geq 0}$, where CId is the set of all coordinator identifiers and E is the set of all events. The coordinator $(cid, vars, initialize, fire, mtds, EQ, t)$ has the identifier cid, the set of variables $vars$, the method $initialize$, the method $fire$, the set of methods $mtds$, the set of events EQ, and the model time t that shows the current time of the model. Each event is an instance of $E = \mathbb{R}_{\geq 0} \times AId \times Msg$, which besides a reference to the receiver actor in a communication, has time information including the time at which the message has been sent. Similar to the actors, the $initialize$ method initializes variables of the coordinator and the main computation of the coordinator is defined in its $fire$ method. The set of methods $mtds$ are defined for planning and analysis purposes.

Similarly, $MCAM = (CA, c, CH)$ defines the multiple interactive coordinated actor models in which CA is a set of CAM, c is a coordinator, and CH is a set of channels.

4.2 Operational Semantics of the Coordinated Actor Model

The same as many other real-time models, we present the operational semantics of the coordinated actor model in terms of TTS. The operational semantics of $CAM = (A, c, CH)$ is defined as $\Pi = (S, s_0, Act, \rightarrow, AP, L)$ such that:

- Each state is an instance of $S = (AId \rightarrow (Vars \rightarrow Val) \times \mathbb{N}) \times (ChId \rightarrow Msg^*) \times (Vars \rightarrow Val) \times \mathbb{R}_{\geq 0} \times \mathcal{P}(E)$, where Val is the set of all possible values of the variables. A state maps each actor of the model to its state variables and a program location $(AId \rightarrow (Vars \rightarrow Val) \times \mathbb{N})$. The program location refers to the currently executing statement of the actor. The state also contains contents of the channels $(ChId \rightarrow Msg^*)$, the coordinator variables $(Vars \rightarrow Val)$, the model time $(\mathbb{R}_{\geq 0})$, and the events kept in the buffer of the coordinator $(\mathcal{P}(E))$. For a state $(\texttt{Atrs}, \texttt{ChS}, v_c, t, \texttt{EQ})$, `Atrs` maps each actor to its local state, `ChS` maps each channel to its sequence of messages, v_c is the local state of the coordinator, t is the model time, and `EQ` is the set of events kept in the buffer of the coordinator. The local state of an actor is (v, pl) such that $v : Vars \rightarrow Val$ returns values of state variables of the actor and pl is its program location.
- The values of state variables of the actors and the coordinator in s_0 are set based on the statements used in the $initialize$ methods of the actors and the coordinator. For each actor, its program location is

zero. The buffer of the coordinator is empty unless the actors use the $selfCall$ statement in their $initialize$ methods. Also, the channels have empty queues and the model time is zero.

- The set of actions is defined as $Act = \{ch.get | ch \in ChId\} \cup \{ch.send(msg) | ch \in ChId \wedge msg \in Msg\} \cup \{et, assign, sc, em\}$.
- The transition relation[1] $\rightarrow \subseteq S \times (Act \cup \mathbb{R}_{\geq 0}) \times S$ includes the transitions related to a sequence of activities such as taking the events from the buffer of the coordinator, firing the actors, and progressing in time. Time progress is enabled when the other transitions are disabled. This sequence of activities is performed in the *fire* method of the coordinator. The transitions are defined as follows.

1) **Event-Taking**: This transition is enabled when the coordinator can select an event, whose time is equal to the current time of the model, from its buffer based on a policy. Then the actor, referred to by the selected event, is fired. It means that its program location is updated to the location of the first statement in a sequence of statements. This sequence that is appended by the $endm$ statement is the body of the *fire* method of the actor. The $endm$ statement denotes when firing of the actor terminates. Furthermore, all events that refer to the fired actor and have times equal to the current time of the model are removed from the buffer of the coordinator. This transition is labeled with et.
2) **Actor-Firing**: By firing the actor, each statement of its *fire* method, referred to by the program location, is executed and the program location is updated to the location of the next statement. The statements result in the following transitions.
   Get: This transition is enabled when the triggered actor executes the $ch.get$ statement. As a consequence, the first message in the queue of the channel ch is removed. This transition is labeled with $ch.get$.
   Send: This transition is enabled when the triggered actor executes the $ch.send(msg)$ statement. As a consequence, the message msg is added to the end of the queue of the channel ch. Furthermore, the event (t, ra, msg) is added to the buffer of the coordinator, where t and ra are the current model time and the receiver actor identifier, respectively. This transition is labeled with the $ch.send(msg)$ action.
   selfCall: This transition is enabled when the triggered actor executes the $selfCall(t')$ statement. As a consequence, the event (t', i, ϵ), containing no message (ϵ), is added to the buffer of the coordinator, where i is the identifier of the current executing actor. This transition is labeled with the sc action.
   Assignment: This transition is enabled when the triggered actor uses an assignment statement. As a consequence, the values of the variables of the actor are changed. This transition is labeled with $assign$.
   End-Method: This transition is enabled when firing of the actor terminates. If none of the input channels of the actor have a message, the program location of the actor is updated to zero. Otherwise, it is updated to the location of the first statement in the body of the *fire* method of the actor appended by the $endm$ statement. It means that the actor is fired until it consumes all of its input messages. This transition is labeled with em.
3) **Time-Progress**: This transition is enabled if none of the above transitions are enabled and the buffer of the coordinator is not empty. The coordinator selects an event from its buffer based on a policy and updates the current model time to the time information of the event. This transition is labeled with $d \in \mathbb{R}_{\geq 0}$, where d shows the amount of the time progress.

- AP contains the name of all atomic propositions.
- $L : S \rightarrow 2^{AP}$ is the labeling function that relates a set of atomic propositions to each state. $L(s)$ relates the atomic propositions that are satisfied in the state s.

As mentioned, the derived semantics for the coordinated actor model is based on TTS. In order to use the compositional verification, we need to determine interfaces of TTSs through which they communicate. To this end, the set of actions of a TTS is partitioned into the input and output action sets Act_I and Act_O, respectively. Note that the actions that do not belong to the interface of a TTS are modeled as internal actions. A TTS with the partitioned set of actions $Act = Act_I \cup Act_O \cup \{\tau\}$ is called a TIOTS. To obtain Act_I and Act_O, we define the sets CH_I and CH_O of boundary input and output channels for the coordinated actor model $CAM = (A, c, CH)$, respectively. The channel $ch \in ChId$ is a boundary input channel ($ch \in CH_I$), if an actor $a_i \in A$ has ch as its input channel ($ch \in Ch_{I_i}$) and none of the actors in A has ch as its output channel. Also, $ch \in ChId$ is a boundary output channel ($ch \in CH_O$), if an actor $a_i \in A$ has ch as its output channel ($ch \in Ch_{O_i}$) and none of the actors in A has ch as its input channel. To use the compositional verification, we transform TTS of a coordinated actor model to its corresponding TIOTS by defining $Act_I = \{ch.get | ch \in CH_I \wedge ch.get \in Act\}$, $Act_O = \{ch.send(msg) | ch \in CH_O \wedge msg \in Msg \wedge ch.send(msg) \in Act\}$, and τ as each element of Act that does not belong to Act_I and Act_O. The set Act_I contains actions of the transitions related to receiving messages from the boundary input channels and Act_O contains actions of the transitions related to sending messages over the boundary output channels. Note that the timed transitions do not change to τ in TIOTS. Thus, there are four types of transitions in TIOTSs that are input, output, τ, and timed transitions.

The syntax of the coordinated actor model is inspired by the modeling language of the Ptolemy framework. Also, the semantics of the coordinated actor model is inspired by the Discrete Event (DE) model of computation in Ptolemy. Actors governed by DE communicate via time-stamped events, where events are processed by each actor in a time-stamp order. Events in DE are handled deterministically. In other

1. The SOS rules of the transitions are available in http://rebeca.cs.ru.is/files/TTCSs.zip

words, the scheduler of DE has only one choice to order the events. Unlike DE, the coordinator in a coordinated actor model is given a policy to handle the events and consequently we can have a nondeterministic model.

The coordinated actor model of a TTCS results in a TIOTS with a finite number of states, since execution of the system terminates at some point, when all moving objects have arrived at their destinations. As passage of each moving object through a sub-track in a TTCS takes an amount of time, time is always advanced and there is no trace of execution in the model of a TTCS along which an infinite number of internal transitions are performed during a limited amount of time (TIOTS is Zeno-free). The model has different execution traces, if the coordinator nondeterministically selects an event among a bunch of events with the same time tag (Event-Taking transition). The semantics of the multiple interactive coordinated actor models is similarly derived based on TIOTS, since we can replace firing of a composite actor with the sequence of firings of its constituent actors.

4.3 Parallel Composition of TIOTSs

Since the proposed approach in this paper is based on composing several TIOTSs, we formally define their parallel composition in this section. Two TIOTSs synchronize on their time progresses if progress in time is jointly performed by both involved TIOTSs. Moreover, they synchronize over their input and output actions, if both of them are jointly involved in performing those actions. Let $\Pi_1 = (S_1, s_{0_1}, Act_{I_1}, Act_{O_1}, \rightarrow_1, AP_1, L_1)$ and $\Pi_2 = (S_2, s_{0_2}, Act_{I_2}, Act_{O_2}, \rightarrow_2, AP_2, L_2)$ be two TIOTSs. The actions in the sets $\{ch.get \in Act_{I_1} | ch \in ChId \wedge msg \in Msg \wedge ch.send(msg) \in Act_{O_2}\}$ and $\{ch.send(msg) \in Act_{O_1} | ch \in ChId \wedge msg \in Msg \wedge ch.get \in Act_{I_2}\}$ are called handshaking input and output actions of Π_1, respectively. The same definition is used for Π_2. As can be seen, a handshaking input action of Π_1 (i.e. $ch.get$) corresponds to a handshaking output action of Π_2 (i.e. $ch.send(msg)$) and vice versa. The same argument is valid for the handshaking output and input actions of Π_1 and Π_2, respectively. Both Π_1 and Π_2 are jointly involved in performing the handshaking actions and the corresponding handshaking actions are hidden. The parallel composition of Π_1 and Π_2 is denoted by $\Pi_1 \parallel \Pi_2 = (S \subseteq S_1 \times S_2, (s_{0_1}, s_{0_2}), Act_{I_1} \cup Act_{I_2}, Act_{O_1} \cup Act_{O_2}, \rightarrow, AP_1 \cup AP_2, L)$. To achieve maximum progress for models in their parallel composition, we assume the transitions labeled with the time and handshaking actions have lower priority compared to other transitions, as shown in the following rules. Assume s_1 and s_2 are current states of Π_1 and Π_2 and $(s_1, s_2) \in S$.

1) **Priority over time progress:** If $s_1 \xrightarrow{l} s_1' \in \rightarrow_1$, $s_2 \xrightarrow{d \in \mathbb{R}_{\geq 0}} s_2' \in \rightarrow_2$, and the action l is not a handshaking action, $(s_1, s_2) \xrightarrow{l} (s_1', s_2) \in \rightarrow$ and $(s_1', s_2) \in S$. The same argument is valid for the case of $s_1 \xrightarrow{d \in \mathbb{R}_{\geq 0}} s_1' \in \rightarrow_1$, $s_2 \xrightarrow{l} s_2' \in \rightarrow_2$.
2) **Priority over handshaking action:** If $s_1 \xrightarrow{l_1} s_1' \in \rightarrow_1$, $s_2 \xrightarrow{l_2} s_2' \in \rightarrow_2$, the action l_1 is not a handshaking action, and the action l_2 is a handshaking action, $(s_1, s_2) \xrightarrow{l_1} (s_1', s_2) \in \rightarrow$ and $(s_1', s_2) \in S$. The same argument is valid for the case of $s_2 \xrightarrow{l_1} s_2' \in \rightarrow_2, s_1 \xrightarrow{l_2} s_1' \in \rightarrow_1$. If both l_1 and l_2 are not the handshaking actions, $(s_1, s_2) \xrightarrow{l_1} (s_1', s_2) \in \rightarrow$, $(s_1, s_2) \xrightarrow{l_2} (s_1, s_2') \in \rightarrow$, and $(s_1', s_2), (s_1, s_2') \in S$.
3) **Synchronization on Handshaking actions:** If $s_1 \xrightarrow{l_1} s_1' \in \rightarrow_1$, $s_2 \xrightarrow{l_2} s_2' \in \rightarrow_2$, the actions l_1 and l_2 are the handshaking actions, $l_1 = ch.get$, and $l_2 = ch.send(msg)$, $(s_1, s_2) \xrightarrow{\tau} (s_1', s_2') \in \rightarrow$ and $(s_1', s_2') \in S$. The same argument is valid for $l_2 = ch.get$ and $l_1 = ch.send(msg)$.
4) **Synchronization on Time:** If $s_1 \xrightarrow{d_1 \in \mathbb{R}_{\geq 0}} s_1' \in \rightarrow_1$ and $s_2 \xrightarrow{d_2 \in \mathbb{R}_{\geq 0}} s_2' \in \rightarrow_2$, $(s_1, s_2) \xrightarrow{d \in \mathbb{R}_{\geq 0}} (s_1'', s_2'') \in \rightarrow$ and $(s_1'', s_2'') \in S$. If s_1' has the smallest time value, $(s_1'', s_2'') = (s_1', s_2)$. If s_2' has the smallest time value, $(s_1'', s_2'') = (s_1, s_2')$. Otherwise, $(s_1'', s_2'') = (s_1', s_2')$. If t is the maximum time value between the time values of s_1 and s_2, t' is the minimum time value between the time values of s_1' and s_2', d is equal to $t' - t$.

As shown by Rule.4, both of Π_1 and Π_2 have to synchronize on their time progresses in $\Pi_1 \parallel \Pi_2$. The states s_1' and s_2' have different time values if Π_1 and Π_2 have different amounts of time progresses from their current states. To synchronize Π_1 and Π_2, the composition progresses in time to reach the smallest time between the times of s_1' and s_2'. This time is the global time of $\Pi_1 \parallel \Pi_2$. Assume s_1' has the smallest time value. To know the amount of the next time progress in $\Pi_1 \parallel \Pi_2$, the current state of Π_2 does not change. In other words, we have a timed transition from (s_1, s_2) to (s_1', s_2), shown by Rule.4. Therefore, the maximum time between the times of s_1' and s_2 shows the global time. If Rule.1 is enabled in (s_1', s_2), Π_1 progresses and its transitions have priority over the timed transition of Π_2 ($s_2 \xrightarrow{d_2 \in \mathbb{R}_{\geq 0}} s_2'$). Note that if s_1' and s_2' have the same time values, we have a timed transition from (s_1, s_2) to (s_1', s_2'). Therefore, one of the rules 2 and 3 can be enable in (s_1', s_2'). We call Π_1 and Π_2 composable if their parallel composition does not reach a deadlock. The parallel composition reaches a deadlock when all transitions in s_1 and s_2 are labeled with handshaking actions and Rule.3 in not valid in (s_1, s_2). In other words, Π_1 and Π_2 are not able to be jointly involved in performing the handshaking actions. Furthermore, the parallel composition reaches a deadlock if all transitions in s_1 and s_2 are labeled with time and handshaking actions, respectively. As a consequence, none of the above rules in valid in (s_1, s_2).

4.4 Example of Parallel Composition of Two TIOTSs

In this section, we show an example of composing two TIOTSs. Consider the multiple interactive coordinated actor models designed for Fig. 1(a), in which there is a coordinated actor model for each component of the system. Assume that each sub-track is modeled by a simple actor whose *fire* method contains the sequence of statements $mOb = ch_i.get()$, $selfCall(t + 1)$, $ch_j.send(mOb)$, $mOb = null$. The actor contains four input channels and four output channels; i.e. ch_i and ch_j are an input and an output channel, respectively. Assume that t is the current

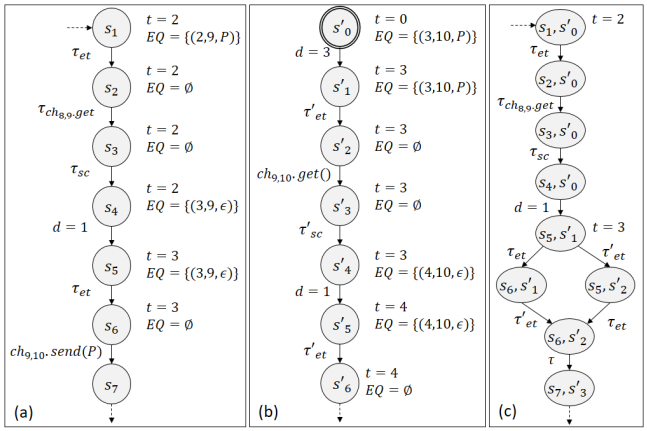

Fig. 2: Parts of TIOTSs of the components C_1 and C_2 of Fig. 1(a) and their compositions are shown in (a),(b), and (c).

model time. The actor receives a message (a moving object), and after one unit of time (as the traveling time of the moving object) sends the message to the next actor. The actor also contains the state variable mOb that shows whether a moving object is passing through the sub-track. The interested reader is referred to [11] to find a more interesting model. Consider that instead of sub-track 8 in Fig. 1, sub-track 14 is unavailable, and the purple aircraft is not rerouted. This aircraft arrives at sub-track 9 at time 2 and enters into the component C_2 at time 3. We abstract away the assignment and the $endm$ statements to draw those parts of TIOTSs of the components C_1 and C_2 in which the sub-track actors 9 and 10 are fired, shown in Fig. 2(a) and Fig. 2(b), respectively. The time and the content of the buffer of the coordinator are shown for each state, and s'_0 is the initial state of C_2. The composition of these TIOTSs is shown in Fig. 2(c). In these figures, P is the message referring to the purple aircraft and $ch_{i,j}$ is the channel connecting two sub-track actors i and j.

5 Compositional Verification of Self-Adaptive TTCSs

In this section, we develop a compositional approach to verify self-adaptive TTCSs in the case of a change occurring and applying adaptation to components. Prior to this, we informally explain our approach on TTCSs. When a TTCS is designed, initial traveling plans of the moving objects are selected in a way that no conflict happens between the moving objects and time properties of the system are satisfied. When a change happens to an area, the moving objects arriving at the area with the unavailable sub-tracks in their routes are rerouted. This way the area is adapted. If the moving objects arrive at the adapted area and depart from it based on their initial traveling plans, the change propagation stops. Otherwise, the change is propagated to the adjacent areas. In this case, all areas affected by the change are composed to form a new area that is adapted. If the moving objects arrive at the new area and depart from it based on their initial traveling plans, the change propagation stops.

We illustrate this approach using an example. Consider a TTCS whose model consists of only two interacting components C_1 and C_2. Let $\Pi_i = (S_i, s_{0_i}, Act_{I_i}, Act_{O_i}, \rightarrow_i, AP_i, L_i)$ be TIOTS of the component C_i. To assure the safe behavior of the system in the absence of a change, it is checked whether Π_1 and Π_2 are composable. None of the safety properties is violated and there is a safe execution for the model if Π_1 and Π_2 are composable. By detecting a change, the component affected by the change is adapted. Consequently, a new TIOTS for the adapted component is obtained. Let $\Pi_{a,i}$ be TIOTS of the adapted component C_i. Suppose that a change in C_1 is detected. If $\Pi_{a,1}$ and Π_2 are composable, 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, in a TTCS with several components, composing TIOTS of the adapted component with TIOTSs of its environment components is an expensive process and may result in a large state space. To reduce the state space, we propose considering the observable parts of the environment components. To this end, we define interface processes for a component. The interface processes of a component (or visible parts of its environment) are TIOTSs of its environment components in which several transitions are hidden. To make a reduced TIOTS, the hidden transitions are removed. In other words, only the transitions related to sending the messages from the environment to the component and vice versa are important in the interface processes. If interface processes of the adapted component and its TIOTS are composable, the change propagation stops. Otherwise, the change is propagated into the environment components.

An interface process is obtained by restricting TIOTS of an environment component to a set of input and output actions. We use $\Pi \downarrow_B$ to show the restriction of TIOTS Π to a set B of actions. The difference between Π and $\Pi \downarrow_B$ is in the label of the transitions. All transitions in Π except for the timed transitions and the transitions labeled with the actions in $\{ch.send(msg)|ch.get \in B\} \cup \{ch.get|ch.send(msg) \in B\}$ are hidden in $\Pi \downarrow_B$. The hidden transitions are labeled with τ. In our example, the interface process of the component C_1 is $\Pi_2 \downarrow_{Act_{I_j} \cup Act_{O_j}}$. To reduce the state space, we fold up a restricted TIOTS by removing its τ transitions. We define a folded TIOTS after defining a trace as follows.

Definition 5.1. *(Trace): A trace from the state s_1 in Π is a sequence of transitions from s_1 to a reachable state s_n, shown by $s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \cdots \xrightarrow{l_{n-1}} s_n$, where $\forall i, 1 \leq i < n, l_i \in (Act_I \cup Act_O \cup \{\tau\} \cup \mathbb{R}_{\geq 0})$. We use $traces_\Pi(s_1)$ to show the set of all finite traces from the state s_1 in Π.*

Definition 5.2. *(Folding a restricted TIOTS): Let $\Pi = (S, s_0, Act_I, Act_O, \rightarrow, AP, L)$ be the restriction of a TIOTS to a set of actions. By folding up Π, the new TIOTS $\Pi' = (S, s_0, Act_I, Act_O, \rightarrow', AP, L)$ is obtained as follows.*

1) *$\forall s_1 \in S, \forall p \in traces_\Pi(s_1) \cdot p = s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \cdots \xrightarrow{l_n} s_{n+1}$ there is $s_1 \xrightarrow{l_n} s_{n+1} \in \rightarrow'$ if and only if $\forall 1 \leq i <$*

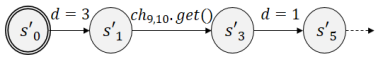

Fig. 3: TIOTS resulted from folding the TIOTS of Fig. 2(b)

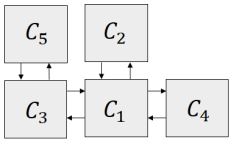

Fig. 4: A model consisting of 5 interactive components

$n \cdot l_i = \tau$ *and* $l_n \neq \tau$.

As described by Rule. 1, Π' contains all transitions of Π except for the τ transitions. The non-reachable states can be removed from the set S in Π'. Note that the defined folding operation preserves the weak timed trace equivalence relation between the original and folded transition systems. Two timed transition systems are in the weak timed trace equivalence relation if and only if they have the same traces, ignoring the τ transitions. We use $\Pi_i \updownarrow_B$ to show that Π_i is restricted to a set B of actions and then is folded. As an example, we restrict TIOTS of Fig. 2(b) to the set $B = \{ch_{9,10}.send(P)\}$ and then fold it. The resulted TIOTS is shown in Fig. 3.

As previously described, if $\Pi_2 \updownarrow_{Act_{I_1} \cup Act_{O_1}} \| \ \Pi_{a,1}$ does not reach a deadlock, the change propagation stops. Otherwise, the change is propagated into C_2. In our approach, by propagating the change from a component to its environment components, we compose the current component with the components affected by the change propagation to create a new component. It is then checked whether the interface processes of the new component and its TIOTS are composable.

The structure of track-based systems enables us to focus on a component and define an interface process 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. For a better understanding, let Fig. 4 shows the multiple interactive coordinated actor models of a TTCS. The interactions are denoted by the arrows between the components. Suppose that a change in the component C_1 of Fig. 4 is detected and this component (its model@runtime) is adapted. If for each component C_i interacting with C_1, $\Pi_i \updownarrow_{Act_{I_1} \cup Act_{O_1}} \| \ \Pi_{a,1}$ reaches a deadlock, the change is propagated into the component C_i. It means that C_1 with its current adaptation is not able to either receive messages from C_i or send messages to C_i. For instance, if $\Pi_3 \updownarrow_{Act_{I_1} \cup Act_{O_1}} \| \ \Pi_{a,1}$ reaches a deadlock, C_1 is not able to either receive messages from C_3 or send messages to C_3. Consequently, C_3 should be adapted to provide inputs for C_1 or receive inputs from C_1 in a different way. Note that to reduce the state space, $\Pi_{a,1}$ can also be restricted to the set of input and output actions of C_i and then be folded.

In the example of Fig. 4, suppose that $\Pi_3 \updownarrow_{Act_{O_1} \cup Act_{I_1}} \| \ \Pi_{a,1} \updownarrow_{Act_{I_3} \cup Act_{O_3}}$ and $\Pi_2 \updownarrow_{Act_{O_1} \cup Act_{I_1}} \| \ \Pi_{a,1} \updownarrow_{Act_{I_2} \cup Act_{O_2}}$ reach a deadlock, and the change is propagated into C_3 and C_2. The components C_1, C_2, and C_3 are adapted and composed to provide the new component $C_{1,2,3}$. If $\Pi_5 \updownarrow_{Act_{O_3} \cup Act_{I_3}} \| \ \Pi_{1,2,3} \updownarrow_{Act_{I_5} \cup Act_{O_5}}$ and $\Pi_4 \updownarrow_{Act_{O_1} \cup Act_{I_1}} \| \ \Pi_{1,2,3} \updownarrow_{Act_{I_4} \cup Act_{O_4}}$ do not reach a deadlock, the change propagation stops, and the change is not propagated further than C_1, C_2, and C_3.

The reason for composing the changed components in our approach is the circular dependency between the components. The change propagated from a component to one of its environment components may back to the component. This means that the component should be adapted again and composability of its corresponding TIOTS and its new interface processes should be checked. This case shows the circular dependency between two components. The compositional verification is not applicable when the circular dependency appears among the components. By composing the components to create a new component, all changes circulating between two components happen inside of the new component and their effects are considered. An example of the circular dependency is shown in Fig. 1, where by propagating the change from C_1 to C_2, the change propagates back to C_1, since the blue aircraft arrives at C_1 at time 7 instead of time 9.

In our approach, the properties of the model are preserved if the interface processes of the adapted component and its TIOTS are composable. However, our approach is inspired by the work of Clarke et al. [8], where each component of the model is supplied with a correctness property and an abstraction of its environment is modeled by interface processes. In [8], by composing a component with its interface processes and verifying a property over the composition, the satisfaction of the property over the whole system is proved. Consider two components P_1 and P_2 with the alphabet Σ_{P_1} and Σ_{P_2}. The restriction of P_1 to Σ_{P_2}, shown by $P_1 \downarrow \Sigma_{P_2}$, is the interface process A_1. Compositional verification using the approach of [8] is formulated as: $A_1 \equiv P_1 \downarrow \Sigma_{P_2} \wedge \psi \in L(\Sigma_{P_2}) \wedge A_1 \parallel P_2 \models \psi \Rightarrow P_1 \parallel P_2 \models \psi$, where L is a logic for reasoning. Unlike this approach, we do not need to define the correctness properties for the components of the model.

6 Implementing and Evaluating the Approach

In this section, we briefly describe the prototype of the proposed compositional approach. This prototype is implemented in Ptolemy II. We exploit an ATC case study to compare the time consumption and the memory consumption between the compositional and non-compositional verification approaches. Ptolemy II is a modeling framework that provides different models of computation with fully deterministic semantics. As mentioned in Section 4.2, the coordinated actor model has a nondeterministic semantics. Therefore, to explore different execution traces, we develop a director with the nondeterministic semantics based on TIOTS and specifically for TTCSs in Ptolemy II. Our director provides the assertion-based verification. It generates the state space of a given component, and performs the reachability analysis. Furthermore, it composes the state spaces of several components, where the components are composed to create a new composite component. The implementation and the evaluation details are described as follows.

6.1 State Space Generation

Algorithm. The algorithm uses Depth-First Search (DFS) and Breadth-First Search (BFS) to generate the state space of a given component. 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. It fires the actor which can be fired in the initial state of the component to generate the next state. The initial state have several outgoing transitions (resp. several next states) if several actors can be fired at the state. The next state is put into the queue if the state is a timed state. Otherwise, the actors which can be fired in the next state are fired. Therefore, the algorithm uses DFS to generate all the traces starting with the initial state and ending with the timed states. It then uses BFS to process the timed states stored into the queue. It dequeues a timed state, and similar to the initial state, uses DFS to generate all the traces starting with that state and ending with the new timed states.

In the case of TTCSs, the algorithm 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.

Composition. Assume that a storm happens at time t, and the component C_1 is affected by the storm. We obtain the state of the real system by estimating positions of the aircraft in the traffic network at time t using their flight plans. We set the initial state of C_1 to the state of the real system. We also set states of boundary actors of the environment components of C_1 to the information of the aircraft traveling through C_1 in the future. These boundary actors will directly send messages to C_1 at times t', $t' \geq t$. To generate the state space, we first compose C_1 with boundary actors of its environment components to create a new composed component. Note that this composition is performed in the level of the coordinated actor model. We then use the above algorithm to generate the state space of the new component. If we could generate the state space, the composition of the state spaces of C_1 and the boundary actors is successfully performed. Otherwise, as mentioned in Section 4.3, the composition faces a deadlock. In the latter case, assume that a boundary actor of 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 boundary 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 boundary 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.

6.2 Experimental Setting

For the comparison purpose, 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 [19], 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. However, 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 does not have the same x or y with 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 a 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). As can be seen, 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 (that 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. However, ALG2 is conscious to not select a stormy sub-track as a part of its route.

Scenarios. We perform two sets of experiments; (ES1) that is to compare the time and memory consumptions, and (ES2) that is to compare the scalability of the approaches. We consider a 9×9 and a 15×15 mesh structure as our traffic networks in (ES1) and (ES2), respectively. These meshes are respectively divided into 9 regions of 3×3 and 9 regions of 5×5. We assume that the fuel of each aircraft is more than the length of the longest path in the traveling network, and is set to 200 and 325 in (ES1) and (ES2), respectively. The threshold of the analysis time is an hour. We also assume that a storm happens in the middlemost sub-track of the networks. In (ES1), we use ALG1 to generate 40 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. Each batch contains flight plans of 1000 aircraft. Per each batch $P_i, 1 \leq i \leq 40$, we generate 10 batches $P_{ij}, 1 \leq j \leq 10$, of flight plans, such that P_{i1} contains the first 100 flight plans of P_i, P_{i2} contains the first 200 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 $\{100, 200, 400, 600, 800\}$. Note that the analysis time of each selected batch is less then an hour. In our experiments, per each j, we average the analysis time and number of the states of the batches P_{ij}.

In (ES2), we use ALG1 to generate a batch P of 7000 flight planes with $\lambda = 0.5$. We assume that the change happens at time 100. In (ES2), 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. Note that an approach is not scalable if by increasing the number of flight plans its analysis time passes an hour.

In our experiments, we assume that FD as the traveling time of an aircraft across a sub-track is 1.

6.3 Comparison

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. We change the network traffic volume and subsequently the concurrency contained in the model through different λs (the inverse of the mean interval time between two departures).

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 in (ES1) are shown in Figures 5 and 6. In this diagrams, "C" refers to the compositional approach and "NC" refers to the non-compositional approach. As shown, the compositional approach decreases the model checking time and the number of states. As expected, by increasing the number of aircraft and decreasing the time at which the storm happens, the number of states and accordingly the model checking time increase. For instance, as a few number of the aircraft have arrived at their destinations at time 100, the number of states at this time is large. More diagrams to compare the model checking time are available[2].

The results of our experiments in (ES2) are shown in Fig. 7. In this scenario, the non-compositional approach fails to model check a model with more than 1000 aircraft in less than an hour. Based on our investigations, the fluctuations appeared between 1600 to 2200 aircraft and also between 3200 to 3600 aircraft indicate that the aircraft added to the traffic network influences on the new routes selected for the moving objects. In other words, in the presence of the new aircraft, the traffic network is adapted in a different way somehow the number of states decreases.

7 Related Work

In this section, the most closely related work are introduced. In [20], an Assume-Guarantee based approach for verifying self-adaptive systems at design time is proposed. In [20], the changed component is adapted. Then, a backward reasoning starts and re-generates 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 [9] 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 Timed Input/Output Automata (TIOA). 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. Unlike [20] and [9], which define an assumption and a guarantee for each component, we only define the interface processes.

Magnifying-Lens Abstraction (MLA), presented in [21], 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 processes and are re-generated by adapting the components. The failure propagation is studied in [22] that checks whether the structural adaptation of the system is fast enough to prevent a hazard. Each adaptation takes an amount of time during which the failure is propagated through the system. After an adaptation, it is checked whether the remaining failures in the system lead to a hazard. Although we do not consider the latency of an adaptation, besides detecting a hazard, we assure that

2. The implementation code and evaluation scenarios are available in http://rebeca.cs.ru.is/files/TTCSs.zip

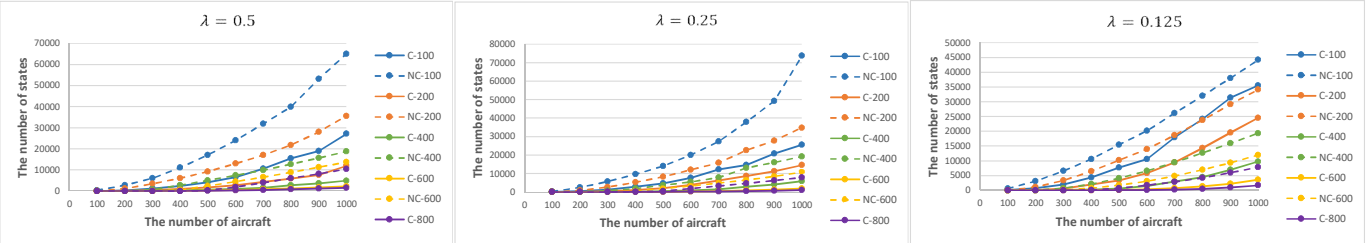

Fig. 5: The number of states in (ES1) for each λ in $\{0.5, 0.25, 0.125\}$

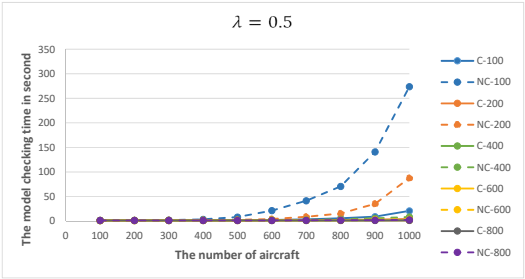

Fig. 6: The model checking time in (ES1) for $\lambda = 0.5$

the time properties of the system are satisfied. The latency-aware adaptation is studied in [6], where a probabilistic model checker proactively selects an adaptation strategy to maximizes the utility of the system. Unlike [6], our focus is on effectively verifying the system behavior.

The works [1], [2], [4], [5], [6], [23] use state-based models (that are defined by states and transitions) to verify self-adaptive systems. Incremental runtime verification of MDPs, described in the PRISM language, is proposed in [2], 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. It then is verified using an incremental verification technique. Runtime verification of parametric Discrete Time Markov Chains (DTMCs) is accomplished in [1]. 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 [23], 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, a self-adaptive framework for runtime verification of self-adaptive systems, is proposed in [5]. RINGA uses Finite State Machines (FSM) to develop a design-time model of a self-adaptive system, and abstracts the model for using at runtime. Each state of the model implements a module of the system, while a transition triggers an adaptation. Each transition is assigned an equation that is parametrized by the environmental variables. The value of the equation is calculated at runtime. Lotus@runtime [4] is a tool for verifying self-adaptive systems at runtime. It uses Probabilistic Labeled Transition Systems (PLTS) to develop a model@runtime. It monitors the generated execution traces of the system and updates the probabilities in PLTS. The desirable properties in [4] are explained through a source state, a target state, a condition to be satisfied, and the probability of satisfying the constraints.

In comparison with the 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 our problem domain applications (there is a one-to-one mapping between elements of the system and the model), facilitates modular analysis of the system. In other words, not only the model of the whole system is decomposed into a set of components, but also the actor model of each component can be decomposed into the smaller components if the analysis yet suffers from the state-space explosion.

8 Conclusion and Future Work

We proposed an approach in which the model of the system is decomposed into a set of components. Upon encountering a change, the component affected by the change is adapted. If the adapted component and its interface processes are composable, the change does not propagate. Otherwise, the environment components affected by the change propagation are adapted. Then, all components affected by the change are composed to create a new component. The same process is repeated for the new component. We model each component by a coordinated actor model, whose semantics is defined based on TIOTS. We use the reduced version of TIOTSs in checking the composability, where only the transitions related to sending messages from the environment to the component and vice versa are considered.

Unlike common compositional verification approaches such as [8], we do not define correctness properties for components of the model. A property of the model is preserved if the adapted component and its interface processes are composable. When a change happens to a component, different adaptation policies are investigated. An adaptation policy may contain the change, and the change does not propagate to the environment components. Therefore, it is possible that the change propagation stops at a time in the future. However, in the worse case that the change propagates to the whole system, all components are composed in our approach. We implemented our approach in the Ptolemy II framework. The results of our experiments confirm that our approach decreases the model checking time and the number of states. To improve our approach, we will study the system behavior in several time windows

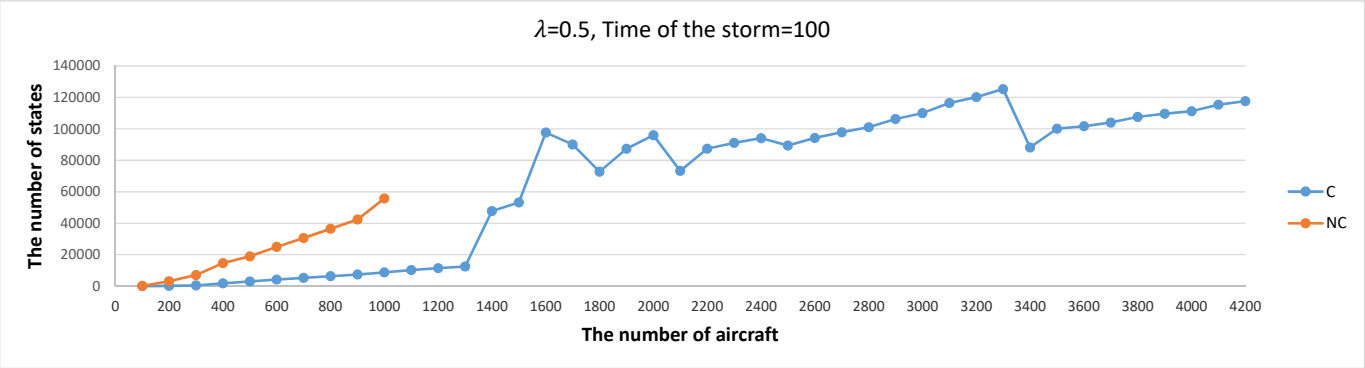

Fig. 7: The scalability of our approach (C) compared to the non-compositional approach (NC)

in the case of propagating the change through the whole system as our future work.

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 authors would like to thank professor Edward A. Lee for his valuable suggestions.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 1–12.

[7] A. Pnueli, *In Transition From Global to Modular Temporal Reasoning about Programs*, 1985, pp. 123–144.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[16] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Form. Methods Syst. Des.*, vol. 34, no. 3, pp. 238–304, Jun. 2009.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

PLACE PHOTO HERE

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 in Sharif University of Technology.

Marjan Sirjani is a Professor and chair of Software Engineering at Mlardalen 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, compositional verification theories, and state-space reduction techniques for actor-based models. She has been working on analyzing actors since 2001 using the modeling language Rebeca.

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.

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 Department of Computer Engineering, Sharif University of Technology, in Tehran, Iran and has been on the Sharif faculty since 1993. His research interests include performance/dependability modeling and formal verification of wireless networks and distributed real-time systems. He is a senior member of the IEEE and the ACM.