HPobSAM for Modeling and Analyzing IT Ecosystems - Through a Case Study

Narges Khakpour[b,a,c], Saeed Jalili[a], Marjan Sirjani[d,e], Ursula Goltz[b], Bahareh Abolhasanzadeh[a]

[a] *Tarbiat Modares University, Tehran, Iran*
[b] *Technical University of Braunschweig, Braunschweig, Germany*
[c] *Leiden Institute of Advanced Computer Science, Leiden, The Netherlands*
[d] *Reykjavk University, Reykjavk, Iceland*
[e] *University of Tehran and IPM, Tehran, Iran*

Abstract

The next generation of software systems includes systems composed of a large number of distributed, decentralized, autonomous, interacting, cooperating, organically grown, heterogeneous, and continually evolving subsystems, which we call IT Ecosystems. Clearly, we need novel models and approaches to design and develop such systems which can tackle the long-term evolution and complexity problems. In this paper, our framework to model IT-Ecosystms is a combination of centralized control and decentralized (self-organizing) approaches. We use a flexible formal model, HPobSAM, that supports both behavioral and structural adaptation/evolution. We use a detailed, close to real-life, case study of a smart airport to show how we can use HPobSAM in modeling, analyzing and developing an IT Ecosystem. We provide an executable formal specification of the model in Maude, and use LTL model checking and bounded state space search provided by Maude to analyze the model. We develop a prototype of our case study designed by HPobSAM using Java and Ponder2. Due to the complexity of the model, we can not check all properties at design time using Maude. We propose a new approach for run-time verification of our case study, and check different types of properties which we could not verify using model checking. As our model uses dynamic policies to control the behavior of systems which can be modified at runtime, it provides us a suitable capability to react to the property violation by modification of policies.

Email addresses: `nkhakpour@modares.ac.ir` (Narges Khakpour), `sjalili@modares.ac.ir` (Saeed Jalili), `marjan@ru.is` (Marjan Sirjani), `goltz@ips.cs.tu-bs.de` (Ursula Goltz), `bahar.abolhasanzadeh@modares.ac.ir` (Bahareh Abolhasanzadeh)

1. Introduction

The next generation of software systems includes complex systems of systems where the individual systems and components are modeled, built, operated, and controlled by different stakeholders, across organizations. Furthermore, software systems and components are equipped with increasing autonomy, including capabilities for self-configuration and self-organization. We call such systems IT Ecosystems. Such software-intensive IT systems can no longer be designed in a purely centralized fashion. Novel approaches are required to design, develop and analyze these systems.

Motivation IT Ecosystems must have the ability to continually evolve and grow even in situations that are unknown during the development time. Due to the fact that it is impossible to fully and properly predict adaptive needs during the design time, adaptive behavior must be built in a way that is *flexible* and modifiable at runtime, because hard-coded mechanisms make tuning and adapting long-run systems complicated.

While each subsystem of a system evolves and changes autonomously to be able to adapt to potentially changing environmental conditions and constraints, they are also cooperating to fulfill a global goal. The centralized control approach to design, in which the behavior of the system is controlled in a top-down way, has attained its limit. In contrast, the decentralized approach relying on a self-organized bottom-up establishment of the desired behavior appears to be infeasible, since we have to make sure that this decentralized approach does not result in unanticipated and undesired behavior. As often, the design of the system has to follow an approach in the middle, somewhere in-between a centralized and a decentralized architecture.

Furthermore, since a complex software system often has a great degree of autonomy, it is more difficult to ensure that it behaves as intended and avoids undesirable behavior. Therefore, to guarantee the functionality of a complex IT Ecosystem, we have to provide mechanisms to ensure that the system is operating correctly, where *model-driven approaches* and *formal methods* can play a key role. Therefore, *we need novel models and approaches to design and develop such systems which can tackle the long-term evolution, flexibility, complexity and assurance problems.*

Different frameworks and models have been introduced to design large-scale software systems inspired by natural systems [1, 2, 3, 4]. Furthermore, [5] proposed a flexible policy-based approach for designing ubiquitous systems. Although most of the existing models are able to exhibit properties of self-organization, self-adaptability, and of long-lasting evolvability, they are not provided with a formal foundation. Also, researchers have paid a lot of attention to formal specification and analysis of dynamic adaptation [6, 7, 8, 9, 10]. Here, most of the existing approaches deal with either behavioral adaptation or structural adaptation [11, 6]. Adaptation, self-* properties, and autonomous computing are however restricted to responding short-term changes, while systems must be additionally able to evolve and grow to cover the long-term evolution of systems [12].

Contribution In this paper we study HPobSAM as a framework for modeling, developing and verifying IT Ecosystem [9, 13, 14] illustrated through a transportation service of a smart airport. Our contributions are as follows:

- We illustrate how HPobSAM can be used as a flexible model for designing IT Ecosystems using a transportation service case study.
- We provide an executable formal specification of the model in Maude [15]. LTL model checking and bounded state space search are used to analyze the model. We found a cross deadlock and the robot collision in our transportation scenario.
- Due to the complexity of our models, we face state explosion problems when we use model checking to verify some properties. Thus, we employ run-time verification [16, 17] as a complement to model checking in which the executions of the system are monitored and checked against a set of formal specifications. We present a new flexible trace-based approach to verify the system at runtime in which properties to be monitored are specified using an algebra. Then, we transform the algebraic properties into a set of policies which are assigned to an observer. A policy expresses whether an event is expected to occur or not. An observer is modeled as a PobSAM manager that uses the policies to check the conformance of the system behavior to the properties.
- In run-time verification, reaction to the property violations is a main challenge. We address this problem by dynamically defining policies to react to the violations.
- To evaluate the applicability of our approach in practice, we have developed a prototype of our scenario using Java and the Ponder2 tool set [18]. We use PonderTalk as the policy language to specify policies.

PobSAM (Policy-based Self-Adaptive Model) [9, 19] is a flexible formal model for developing and modeling self-adaptive systems which uses policies as the principal paradigm to govern and adapt the system behavior. Policies are known as a powerful mechanism to achieve flexibility in adaptive and autonomous systems which allow us to *"dynamically"* specify the requirements in terms of high level goals. A PobSAM model is a collection of actors, views, and autonomous managers. The autonomous managers govern the behavior of actors by enforcing suitable policies using contextual information provided by views. This model supports behavioral adaptation through modifying the policies used to control the system behavior introduced in [9]. HPobSAM is an extension of this model to support hierarchical modeling and structural adaptation introduced in [14], in which a manager is aware of its substructure and adapts its substructure to the changing environment according to policies. HPobSAM has a formal foundation that employs an integration of algebraic formalisms and actor-based models. The structural operational semantics of HPobSAM is described using graph transition systems and hierarchical hypergraph transformation. In this paper, we (i) study the applicability of HPobSAM in designing IT Ecosystems, (ii) provide the mapping of HPobSAM to Maude, (iii) propose a new runtime verification approach for HPobSAM, and (iv) introduce the detailed case study of the smart airport and its modeling, verification and implementation using HPobSAM.

In this paper we explain HPobSAM and our analysis techniques through a case study, smart airport. The smart airport case study is introduced in Section 2. After giving an overview of HPobSAM in Section 3, in Section 4 we explain the modeling framework and discuss why HPobSAM is suitable for modeling IT ecosystems in general. In Section 5 we show the HPobSAM model for the smart airport. In Section 6, we present the Maude specification of our model and analyze the model formally. An approach is proposed to verify our case study at run time in Section 7. We compare our approach with related work in Section 8, and Section 9 concludes the paper.

2. Case Study Overview

The airport departure scenario is an example of a software-intensive system of systems [12]. We use a transportation service at the departure area of an airport as our case study. This transportation service is supposed to be realized by a number of Autonomous Transport Vehicles (ATVs) which are responsible to transport passengers between stopovers including passenger entrances, check-in desks, departure gates, and plane parking positions. There are several two-lane roads which connect the aforementioned stopovers. To avoid congestion and blockages, there are some side roads which can be used instead of the main roads (implying a reduced vehicle speed).

There are a variety of ATVs of different sizes to perform transportation in a self-organizing manner. All ATVs know the airport map and stopovers. The transportation service of the transport vehicles contains transporting passenger (i) from an airport entrance to one of the five check-in desks, (ii) from a check-in desk to one of the departure gates, and (iii) from a departure gate to the correct parking position of the respective plane. ATVs consume energy while driving on roads, and they have to recharge their batteries at a charging station.

The observation systems (e.g., smart cameras, sensors, RFID readers) placed around the area gather and provide information (e.g. the current traffic information). This information is used by ATVs in order to achieve a good performance of transportation. Furthermore, passengers use a mobile device, called SmartFolk, to interact with the IT systems at the Smart Airport. A SmartFolk can be seen as a device like a PDA. Within the IT Ecosystem they represent their owners and act as interfaces to the airport IT Ecosystem.

ATVs are signed in a service named transport scheduler that collects passenger orders and offers tickets (pickup/drop positions, times) to the ATVs. Hence ATVs have to collaborate and negotiate in competition on tickets, roads and charging stations. To prevent the occurrence of unsafe situations caused by a selfish acting ATV, we need to implement a mechanism to balance agent autonomy and system controllability.

3. HPobSAM

A HPobSAM model consists of the following elements:

- *Actors* are computational entities dedicated to the functional behavior of the system.
- *Self-Adaptive Modules* (SAM) are the building blocks of a model which are able to automatically adapt their behavior in a complex dynamic environment. A self-adaptive module may be either a federation of self-adaptive modules collaborating to achieve a particular goal, or a composition of self-adaptive modules or actors governed by a manager.
- *Views* provide an abstraction of the state of actors and self-adaptive modules for the managers.
- *Managers* are responsible for managing the behavior of actors and lower-level self-adaptive modules according to the predefined policies. A manager may have different configurations. Each configuration consists of three classes of policies: governing policies, behavioral adaptation policies and structural adaptation policies. A manager uses the governing policies for directing actors and controlling the behavior of lower-level self-adaptive modules (SAMs) and actors by sending messages to them. The behavioral adaptation policies are used for switching among the configurations. The structural adaptation policies are used for changing the structure of SAMs by adding or removing agents.
- *Roles* are notions to group agents with the same functionality where an agent is a self-adaptive module or an actor. The managers' policies are specified in terms of roles, and agents are assigned to the roles dynamically as a means to restructure the system.

The main elements of a HPobSAM model are managers. A manager is defined as a tuple $m = \langle C_m, c_{init}, V_m, H_{\tau_m}, H_m \rangle$, where C_m is the set of m's configurations, $c_{init} \in C_m$ is its initial configuration, and V_m is the set of observable views of m. The typed hierarchical hypergraph [20] H_{τ_m} shows the roles, agent types and their relationships (assignment of roles to the agents). The hypergraph H_m is a H_{τ_m}-typed hierarchical hypergraph that describes how m is connected to other agents, i.e. the agents which the manager has interaction with. A configuration $c \in C_m$ is defined as $c = \langle gp, bp, sp \rangle$, where gp, bp and sp indicate the governing policy set, the behavioral adaptation policy set and the structural adaptation policy set of c, respectively.

Governing Policies. A simple governing policy $gp_i{=}\langle o, e, \psi\rangle \bullet a$, $gp_i \in gp$, consists of priority $o \in \mathbb{N}$, event $e \in E$ where E is an assumed set of possible events, condition ψ (a Boolean term) and an action a. An event is triggered when a specific condition in the system becomes true, when the execution of a message server is completed, when a message is sent, and when a new object is created/removed. The actions in the governing policies are specified using an algebra CA^a defined as follows. We let a, a' denote action terms, while an α is a primitive action.

$$a \overset{def}{=} a; a' \;\mid\; a \parallel a' \;\mid\; a \,\lfloor\!\lfloor\, a' \;\mid\; a + a' \;\mid\; \phi :\rightarrow a \;\mid\; \alpha \;\mid\; \delta_g$$

Thus an action term can be a sequential composition (;), a parallel composition ($\parallel$, $\lfloor\!\lfloor$), a non-deterministic choice (+), or a conditional choice ($\phi :\rightarrow a$). Moreover, we have the special constant δ_g as the deadlock action for governing policies. A primitive action of a simple governing policy is of the forms $r.msg$ to send the message msg to the agents with role r.

Structural Adaptation Policies. A simple structural adaptation policy $sp_i{=}\langle o, e, \psi\rangle \bullet a$, $sp_i \in sp$ consists of priority $o \in \mathbb{N}$, event $e \in E$, condition ψ and an action a which is specified as a CA^a term. A primitive action of a structural adaptation policy is of the forms (*i*) $join(r, \omega)$ for assigning role r to the agent ω, (*ii*) $quit(r, \omega)$ for releasing agent ω from role r, (*iii*) $add(\omega)$ and $remove(\omega)$ for adding and removing an agent, and (*iv*) $r.msg$ to send the message msg to the agents with role r. In other words, each manager is aware of its substructure and responsible for structural adaptation of its corresponding module using structural adaptation policies. Execution of structural adaptation actions leads to transforming H_m into H'_m which contains the structural modifications.

Whenever a manager receives an event e, it identifies all the (governing and structural adaptation) policies that are triggered by that event, i.e. are of the form $\langle o, e, \psi\rangle \bullet a$ for some o, ψ, and a. For each of these activated policies, if the policy condition ψ evaluates to true and there is no other triggered policy with priority higher than o, then action a is executed. Note that behavioral/structural adaptation policies have higher precedence than governing policies.

Behavioral Adaptation Policies. A simple behavioral adaptation policy $bp_i = \langle o, e, \psi, \lambda, \phi\rangle \bullet c$, $bp_i \in bp$ consists of priority $o \in \mathbb{N}$, event $e \in E$, and a condition ψ (a Boolean term) for triggering the adaptation. Moreover, condition ϕ is a Boolean term indicating the conditions for applying the adaptation, λ is the adaptation type (loose, denoted $\perp$, or strict, denoted $\top$), and c is the new configuration. Informally, behavioral adaptation policy $\langle o, e, \psi, \lambda, \phi\rangle \bullet c$ indicates that when event e occurs and the triggering condition ψ holds, if there is no other triggered adaptation policy with priority higher than o, then the manager evolves to the strict or loose adaptation mode as given by λ. When the condition ϕ is true, it will perform adaptation and switch to configuration c. The behavioral adaptation policy of a manager is defined as composition($\oplus$)

of the simple behavioral adaptation policies. Furthermore, δ_p indicates the null behavioral adaptation policy.

The operational semantics of HPobSAM is defined in terms of prioritized conditional graph transition systems [14]. Graph transition systems are essentially classical transition systems augmented with a function mapping states into graphs and transitions into partial hierarchical morphisms. Thus every state is provided with a graph indicating the current system structure.

4. Suitability of HPobSAM for Modeling IT Ecosystems

While subsystems of an IT Ecosystem evolve and change autonomously to adapt to changing environmental conditions and constraints, they are also cooperating to fulfill a global goal. There are various architectural strategies to design such systems [21]. The classical approach is based on a top-down central control of the behavior of a system. This approach has reached its limits. In contrast, the decentralized approach relies on a self-organized bottom-up establishing of the desired behavior. A full bottom-up design appears to be infeasible, since we have to make sure that this decentralized approach does not result in any unanticipated and undesired behavior. As often, the solution will follow an approach in the middle, with the system's design in-between a centralized and a decentralized architecture.

Figure 1 gives a schematic view of an IT Ecosystem, which is decomposed into a set of self-adaptive modules. A self-adaptive module, in turn, may contain a number of self-adaptive modules structured hierarchically. A typical self-adaptive module consists of a general manager and a federation of either self-adaptive modules, or actors, collaborating to achieve a particular goal in a self-organizing manner. In the sequel, we discuss how this model addresses the explained requirements of an approach for designing IT Ecosystems.

Adaptation Support. Adaptation in a self-adaptive module is realized in two ways:

- *Behavioral adaptation* The module's manager is provided by a collection of dynamic policies to control and adapt the behavior of its controlled agents in a centralized manner. The manager controls the behavior of actors through sending them messages. Moreover, it controls and adapts the behavior of self-adaptive modules under its control by changing their managers' configurations.

- *Structural adaptation* The manager is aware of its substructure and performs structural adaptation by adding, removing and replacing the controlled agents and their interconnections.

Incorporating centralized and decentralized design. The managers of self-adaptive modules interact with each other to achieve a specific goal of their immediate higher-level self-adaptive module. Manager interactions are realized by two mechanisms: message passing and shared memory. The view layer acts as the tuple space shared among the managers, where a module shares information with other self-adaptive modules. The behavior of a self-adaptive module is adapted locally by the manager in a centralized way through enforcing policies. Furthermore, self-adaptive modules at the same level of hierarchy collaborate and interact with each other in a self-organizing manner to achieve a higher-level goal. In other words, the control of the system is distributed among the self-adaptive modules. Therefore, the design of the system is somewhere in-between a centralized and a decentralized architecture.

Controlled Autonomy. The ability to change configurations (policies) of a manager by a higher-level manager enhances the controllability of the system, since the self-adaptive modules are not fully autonomous anymore. In other words, although each SAM behaves as autonomous as possible to achieve its goals, this autonomy, however, is partially controlled by the upper-level managers to accomplish the higher-level goals.

Flexibility and Long-term Evolution. Policies are high-level specifications which can be defined and loaded dynamically. The managers interpret the policies and control the system behavior according to them. We can change policies used to control the system behavior at runtime which leads to changing the behavior of system consequently. *Thus, PobSAM allows us to adapt to unforeseen situations without the need to modify the low-level system code by simply defining a new set of policies.* The messages $add(c)$ and $remove(c)$ sent to the manager, are used to add and remove configuration c, respectively. Furthermore, when the manager receives a message $load(c, \lambda, \phi)$, it performs an adaptation for switching to configuration c where λ denotes the adaptation type and ϕ indicates the condition of applying adaptation. Hence, we can simply define and load a new configuration containing new policies in case the system requires to be evolved. Moreover, the manager can be instructed to switch to an already defined configuration to address the long-term evolution. The adaptation logic is specified in terms of adaptation policies which can be modified at runtime. Therefore, the adaptation mechanism is flexible, i.e. we can change policies used for behavioral and structural adaptations dynamically. We can conclude that the policy-based design of systems enhances the flexibility and supports the long-term evolution of systems.

Scalability. Due to the large-scale of IT Ecosystem, scalability is a significant feature of a model to design such systems. HPobSAM allows us to build the system hierarchically from adaptive/evolvable components (i.e. self-adaptive modules), therefore it enhances the scalability of models. Scalability in a centralized approach is limited, because a single control point is responsible for collecting and processing the control information. Since HPobSAM is a decentralized model in terms of control points, control information is collected and processed locally by the manager. Therefore, this model with local control points scales well in terms of size. Moreover, since a manager collects and processes information locally, it improves the performance.

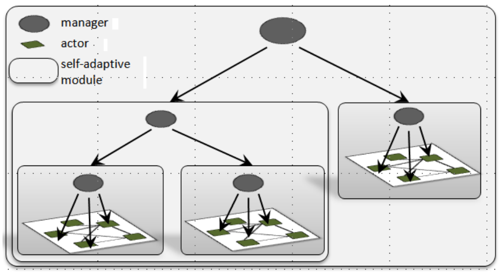

Figure 1: The typical IT Ecosystem

5. Modeling the Transportation Service using HPobSAM

In our scenario, the main subsystems are ATVs with various capabilities, smart folks, the transportation scheduler, charging stations, check-in desks and departure gates. Each subsystem is modeled as a self-adaptive module. Due to the lack of space, we restrict ourselves to introduce the modeling of ATVs.

Modeling Individual ATVs. The hierarchical hypergraph shown in Figure 2 gives the simplified architecture of an ATV self-adaptive module, designed using HPobSAM. This is done in a simplified way while abstracting from technical issues. We draw managers as circles, actors as small square boxes, self-adaptive modules as double-lined square boxes, and roles as rounded rectangles. An ATV self-adaptive module includes a top-level manager called *ATV Controller* and the self-adaptive modules *Path Planner Module*(PM), *Energy Module*(EM), *Motion Module*(MM), *Brake Module*(BM), and *Task Module*. Figure 3 presents the PobSAM model of an ATV partially. The manager *ATV Controller* has a configuration named `normalConf` to control the ATV in normal conditions. The set of governing policies in configuration `normalConf` is `{ngpA,ngpB,ngpC}`. Moreover, the behavioral adaptation policy `napA` is used to switch to configuration `collisionConf` when a collision occurs. No structural adaptation policy is defined for this configuration. The configuration `collisionConf` is defined to control the robot in case of collision.

The self-adaptive module *Path Planner Module* consists of (i) a manager whose governing policies determine the best path based on the state of the robot, energy level, traffic info etc, and (ii) two actors for computing the shortest path in terms of distance or time. *Motion Module* is a self-adaptive module whose manager is responsible to control the movement of the robot to a target using a set of policies. For instance, `ngpAM` states that when the motion controller receives a signal for moving to `target`, it first asks *PathPlanner* to find the best path to the target in the current context. Given the best path, *Motion module* directs the robot by controlling velocity and direction of the robot through the actor *EnginControl*. The ATV moves towards the target with the specified speed until it receives a message to decrease or increase the speed, or needs to brake for collision avoidance. Moreover, when a robot arrives at a cross, it may

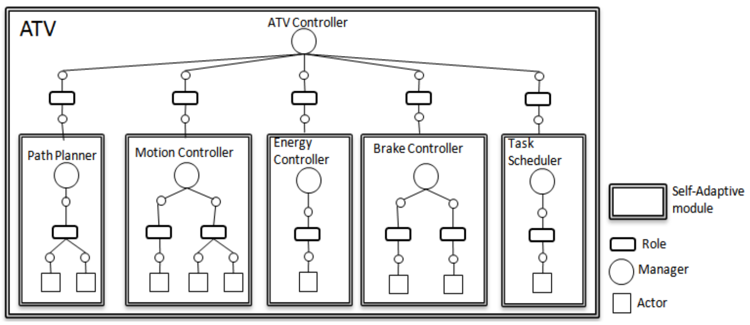

Figure 2: The Hierarchical Hypergraph of an ATV

recompute the best path, since it is likely that the current path is not the best path anymore due to changing roads' traffic load, accident etc (`ngpBM`).

The self-adaptive module *Brake Module* is in charge of adaptive braking of the robot. As an instance, a policy asserts that when an obstacle is detected in a short distance, the robot should brake to prevent collision. The *EnergyModule* is a self-adaptive module responsible for energy management and charging the robot. When the robot's energy decreases to a specific level, a message is sent to the motion controller for finding the nearest charging station to recharge the battery. When an ATV arrives at the charging station, it receives a number indicating its position in the charging queue. The actor of this module is *battery* which is provided with a sensor indicating the energy level.

ATVs compete to transport passengers as their main task. We consider the self-adaptive module *TaskScheduler* with a manager to negotiate with other robots and schedule tasks effectively. The actor *Taskinfo* of this module maintains information about the tasks transported by the robot. The view layer of an ATV contains its current position, intermediary target, final target, status (normal, urgent, emergency), energy level, map with traffic info, last map used to compute path, number of transported tasks, current task etc.

```
MANAGERS{
 MANAGER ATVController
 {
  statevars {
     byte ATVId;
      }
  managedElements{
    PathPlannerModule PP;         MotionModule MM;
    EnergyModule EM;       BrakeModule BM;
    TaskScheduler TS;
   }
   roles {
    //definition of roles
  }
 configurations{
    normalConf=[nbpA] [ngpA,ngpB,ngpC] [];
    collisionConf=[cbpA] [cgpA,cgpB] [cspA];
    }
```

```
policies{
   nbpA : on collision if true switchto collisionConf when true priority 1;
   //definition of policies
  }
views {
    byte ATVspeedT = MM.ATVspeed;
    //definition of other views
   }
}
MANAGER MotionController
  {
    statevars {
     }
    managedElements{
      EngineControl engineControl;
      PathPlanner   pathplanner;
     }
    roles {
       //definition of roles
    }
    configurations{
        normalConfM=[nbpAM,nbpBM] [ngpAM,ngpBM,ngpCM][];
        idleConfM=[ibpAM,ibpBM] [igpAM,igpBM] [];
        collisionConfM = [cbpAM,cbpBM] [cgpAM,cgpBM] [cspA];
      }
    policies{
        ngpAM : on start(target,status) if true do
            pathplanner.findpath(target,status);
            <status==normal:->engineControl.setspeed(normal)> +
            <status==emergency:->engineControl.setspeed(high)> +
            <status==urgent:->engineControl.setspeed(high)> +
            engineControl.start() priority 1;
        ngpBM : on oncross() if newtrafficload =/= oldtrafficload do
            pathplanner.findpath(target,status);
            engineControl.start() priority 1;
        //definition of policies
     }
    views {
     byte ATVspeed = engineControl.speed;
     //definition of other views
    } }
 //definition of the rest of managers
 }
ACTORS {
   reactiveclass EngineControl() {
      knownrebecs {}
      statevars{public byte Speed; }
      msgsrv moveforward()  {
        ...
       }
      msgsrv incSpeed()  {
        ...
       }
     //definition of the rest of message servers
       }
    //definition of the rest of actors
 }
SAMS{
 SAM PathPlannerModule {
      PathPlanner PP(TPC,DPC);
      TimePathComp TPC();
     DisPathComp  DPC(); }
SAM MotionModule {
     EngineControl EC;
     MotionController MC(EC,PP); }
//definition of the rest of SAMs
}
```

Figure 3: The PobSAM specification of an ATV

Modeling Collaborations. Thus far we have presented a centralized approach to design an ATV architecture. Now we concentrate on interactions and collaborations between ATVs and other subsystems. ATVs need to interact and collaborate with each other, the transportation scheduler, charging stations, check-in desks and departure gates. They require to have direct access to information provided by their vicinity. As an example, consider the simple governing policy of an individual ATV defined for passing a junction: "Move forward when the robot is on a junction and has a higher priority than the other ATVs at the crossing". Here, the ATV needs to know the priority of other ATVs at the junction defined in terms of their energy level, job priority, distance to the target etc. The view layer acts as the tuple space shared among ATVs to coordinate their interactions, and an ATV can have controlled access to the view layer of other ATVs in its locality. Thus, an ATV can obtain neighbourhood information through a common view layer shared among ATVs in a locality. The idea of a common view layer between ATVs is similar to the notion of virtual environment introduced in [22].

Moreover, ATVs require to perceive properties of the global system state, specially in situations such as congestion condition when ATVs' information is insufficient for vehicles to determine the best path. Hence we partition the smart airport area into smaller regions called *cells*, where a cell contains an autonomous device, called *cell controller*, deployed within the cell physically. A cell controller is specified as a manager which is aware of the ATVs and other subsystems located in its defined physical area. Figure 4 shows a simplified specification of a cell controller. A cell controller with its governed agents, including ATVs, smart cameras, check-in desks, departure gates, and charging stations form a self-adaptive module. The main responsibilities of a cell controller are: (i) providing ATVs with necessary global state information (ii) changing ATVs' configurations dynamically to control their full autonomy, and (iii) notifying other cell controllers of changes which influence their cells (e.g. congestion condition). Thus ATVs are adaptive in the sense that they can change their configurations dynamically to adapt to different contexts, however the cell controller can also define new configurations for ATVs to balance between controllability and autonomy of agents (e.g. see governing policy `ngp1` of the cell controller).

```
MANAGER CellController
 {
  statevars {
         byte CellId;
        }
  managedElements {
      ATV atvA;     ATV atvB;
      //definition of the rest ATVs
      smartCamera cameraA;
      chargingStation cs1;
      CellController adjacentA; CellController adjacentB;
        }
  roles {
      LATVs = {atvA,atvB} ;
      smartCamera = {camera1} ;
      adjacentCells = {adjacentA,adjacentB};
      //definition of the rest roles and role assignments
      allATV = {atvA,atvB,...};
        }
  configurations{
                 //the cell controller only has one configuration
                 normalConf=[apnone] [ngp1] [nsp1,nsp2,nsp3];
               }
  policies {
            //governing policies
            ngp1: on congestion(x,y) if true do
                       forall atv in allATV,cell in adjacentCells
                        atv.loadconfig(newconf) ||
                        cell.inform(congestion(x,y))
                 priority 1 ;
            // structural adaptation policies
            nsp1: on LATVReq(cl) if handleable(reqNo,[0,0,1]) do
                       forall atv in LATVs
                        (atv.isfree and atv.isnearest(cl)) :->
                          (atv.moveto(loc);atv.joinReq(cl))
                    priority 1;
            nsp2: on onLeave(atv) if true do remove(atv) priority 1;
            nsp3: on onJoin(atv)  if atv.hasEnergy and H do
                       add(atv);join(LATVs,atv)
                    priority 1;
           //definition of policies
      }
  views {
          byte atvAspeed = atvA.ATVspeed;
          byte atvAPosX = atvA.ATVPosX;
          //definition of other views
     }
}
```

Figure 4: The PobSAM specification of a cell controller

Structural Adaptation. We illustrate self-organization and structural adaptations in our scenario using an example. Suppose there are two types of ATVs with different capacities to transport passengers. Several large groups of travelers arrive at the airport's entrance. When the growing crowd of travelers is noticed by the corresponding cell controller, it notifies its adjacent cells about the need for more ATVs. A decentralized adaptation is carried out by the cell controllers to adapt the system to the current context. The policies `nsp1,nsp2` and `nsp3` are defined for the cell controllers to handle this situation. Policy `nsp1` states that on request of a large ATV from cell controller `cl`, if the existing requests of the cell can be handled by sending a large ATV to `cl`, a free large ATV which is nearest to `cl` is sent to this cell. The sent robot requests the `cl`'s cell controller to join its cell. After joining `cl`, it is removed from the list of provider's large ATVs (policy `nsp2`). When a new ATV requests to join a cell, the corresponding cell controller checks the required capabilities of the requester for acting as an equipped ATV, i.e. having large capacity, video camera, powerful motor and enough battery using policy `nsp3`, in which `H` defines the constraints on the ATV structure. Then role LATV will be assigned to the new ATV by the cell controller. Notice that policies `nsp1` and `nsp2` are executed by the provider cell controller, and the requester's cell controller enforces policy `nsp3`.

Summary of the Section. A centralized approach is used to design an ATV where the managers (i.e. $ATVcontroller$s, $MotionContoller$s, *EnergyController* etc) control the behavior of different parts of the ATV (i.e. *EngineControl*, *Brake*, *Battery* etc) using dynamic policies autonomously. ATVs collaborate with each other in form of self-organization to achieve the high-level goals, i.e. by performing distributed structural and behavioral adaptations. The cell controllers control this self-organization in a centralized manner using policies by changing the ATVs' configurations.

6. Model Checking

In this section, we present an executable specification of our model in Maude. We use Maude analysis tools including its LTL model checker and search commands to analyze the model of our case study formally.

6.1. Maude Overview

Maude is a formal language and tool set based on rewriting logic used for developing, prototyping, and analyzing formal specifications. Intuitively, states of a system are represented as elements of an algebraic data type in rewriting logic, and computation is given by local transitions between states described by rewrite rules. A rewrite rule is of the form "`t => t' if c`" where `t` and `t'` are terms representing a substate of the system, and `c` is a condition on the variables of `t`. This rule says that when the system has a subcomponent matching `t`, and the rule condition evaluates to true, that subcomponent can evolve to `t'` through replacing by the rule right-hand side. The application of a rule is possibly done concurrently with changes described by rules matching other parts of the system state. The process of application of rewrite rules generates computations. A computation is a possibly infinite sequence of rewrites $S_0 \xrightarrow{l_1} .... \xrightarrow{l_n} S_n$ where l_i is a label determined by the rewrite rule applied and S_i for $0 \leq i \leq n$ indicates the system state.

Maude is provided with an LTL model checker which allows us to check whether every possible behavior starting from a given initial model satisfies a given LTL property. LTL is a temporal logic that extends propositional logic by introducing four basic temporal operators: the existential operator `<>`, the global operator `[]`, the until operator `U` , and the next operator `O`. An LTL formula is evaluated over a sequence of states. Given formulas ϕ and ψ, `[]`ϕ (read as always ϕ) states that ϕ is true in all states. `<>`ϕ (read as eventually ϕ) means ϕ is eventually true. ϕ `U` ψ (read as ϕ until ψ) denotes ϕ is true until ψ becomes true. `O`ϕ (read as next ϕ) asserts ϕ is true in the next state.

6.2. Specification and Analysis of the Model

The time and feasibility of our analysis depends on the size and complexity of the system model. Our system model is not directly amenable to analysis due to its complexity. We use a number of techniques to manage the complexity of our model:

- *Data Abstraction* Data abstraction is a technique to reduce the state space which focuses on finding a mapping between the concrete values of the system variables and an abstract set of variables representing the original data values using a homomorphic abstraction function.
- *Behavior Symmetry* We can take advantage of the symmetric behavior of self-adaptive modules and prove the property for a minimum number of self adaptive modules. The required number of self-adaptive modules depends on the property to be checked. The modeler must be careful about using this technique to reduce verification complexity.
- *Bounded Reachability Analysis* The Maude `search` command allows us to explore the reachable state space following a breadth-first strategy. This can be used to find reachable states satisfying a user-defined property. The idea of bounded search is that we check a property, not for all reachable states, but only for those states reachable within a certain depth bound.

Given the executable specification of a model in Maude, it can be used to simulate and analyze the system. We will focus on three kinds of analysis: (i) simulation, to execute the system specification; (ii) reachability analysis, to look for deadlocks and policy conflict detection, and to prove system invariants; and (iii) LTL model checking, to analyze those safety and liveness properties which can not be specified using reachability analysis (`search` command). Furthermore, we have built a tool to generate Maude specification from a PobSAM model automatically. Although we can produce the Maude specification of managers and self-adaptive modules completely automatically, the actors specification is produced only partially and the user has to complete the specification of message servers manually.

Simulation. Maude specifications can be executed using the `rewrite` and `frewrite` commands, which implement two different execution strategies: a top-down rule-fair strategy, and a depth-first position-fair strategy, respectively. Thus, we can execute the model by simply typing `rewrite initModel` where `initModel` is a term representing the initial state of the model. We can also specify upper bounds for the number of rule applications in the `rewrite` and `frewrite` commands. This can be very useful to perform step-by-step executions.

Conflict/ Anomaly Type	**Description**
Action Conflict	Two policies with conflicting actions are triggered simultaneously
Effect Inference Conflict	Enforcement of a policy overrides the effect of another policy
Inexecutable Action Conflict	Enforcement of a policy makes performing the action of another policy inexecutable by violating its prerequisites
Unenforceable Policy	A policy will never become activated because its triggering conditions never become true

Table 1: Policy Conflicts Types

Formal Analysis. In general, properties to be checked about an adaptive system can be categorized as adaptation properties, functional properties or composition of both. We discuss common properties which can be checked for all applications. It is clear we also need to verify application-specific properties as well. As policies direct the system behavior in our model, it is required to understand and control the overall effect of governing policies on the system behavior. A policy may not be enforced properly due to conflict with other policies, i.e. a triggered policy can only be enforced correctly provided that it has no conflict with other policies. Two policies ρ_i and ρ_j are in conflict if (i) simultaneous activation of policies ρ_i and ρ_j leads to a state wherein the system cannot choose a policy to enforce, (ii) applying ρ_i leads to a situation which makes executing the action of ρ_j impossible, (iii) enforcing ρ_i and ρ_j (that not necessarily become enabled simultaneously) results in executing two conflicting actions. We say two actions are in conflict if either execution of an action violates the effect of the other action or satisfying the post-conditions of both actions is infeasible due to the constraints of the system.

Table 1 gives a taxonomy of conflicts which may exist among governing policies (presented in [19]). We use the `search` command to detect action conflicts, inexecutable actions and unenforceable policies, and LTL model checking is employed to detect the effect inference conflicts. The `search` command for detecting action conflicts searches for a state where two triggered policies with conflicting actions are in the triggered policy list of the manager.

Regarding adaptation properties, when a manager is in the adaptation mode, we should verify that it would eventually switch to the next configuration. The LTL formula `~<>[] C` is used to verify this property where `C` is a term denoting that the manager is in the adaptation mode. Moreover, we can check whether all configurations of a manager can be reached finally or not. If searching the state space leads to finding a state wherein the current configuration of the manager (`curconfig`) is `c`, we say `c` is reachable. Furthermore, deadlock-freedom is a generic property of a system that must be checked.

6.3. Analysis of Transportation Service

We use an abstract model of our case study for verification purpose. Abstract values are considered for attributes such as energy which ranges over real domain, e.g. we reduce the energy values to the abstract domain {low, normal, high}. We also take into account an abstract layout of the airport map with one entrance, one check-in desk, one junction, and one charging station. It is assumed that all ATVs are identical with the same configuration sets. To make the model checking procedure more efficient, we take advantage of the symmetric behavior of ATVs. Regarding the properties that we are interested to verify in our case study, two ATVs are considered in our model. As the number of properties to be checked is high, we do not try to produce a complete list of properties. We restrict ourselves to give a number of properties checked in the sequel.

Action Conflict. One of the reasons of action conflicts is presence of inconsistency in determining the next direction of ATV, i.e. there are different governing policies instructing the ATV to move to different directions. The following command searches for a state such as `confState` from `initState` within depth 1000, where `tp?` denotes the triggered policies of the motion controller of `ATVA` in state `confState`. Furthermore, the governing policies `ngpA(ATVA)` and `ngpE(ATVA)` have conflicting actions `moveForward` and `moveRight`, and the function `memberof` checks whether a policy is in the list `tp?` or not.

```
search [1,1000] initState =>* confState such that
           (memberof(tp?,ngpA(ATVA)) and memberof(tp?,ngpE(ATVA))).
```

Inexecutable Action. An ATV requires a minimum level of energy to transport passengers to a target, otherwise it will become out of energy which causes traffic jam. We search for reachable states such as `trgState` where the action `setTarget` is the head of input queue of the ATV and the robot does not have enough energy.

```
search [10 , 2000] initState =>* trgState
                such that (energyval = low ) .
```

Unenforceable Policy. The governing policy `gp` is enforced eventually, if we can find a state wherein this policy belongs to the set of triggered governing policies:

```
search [1,1000] initState =>* trgState
              such that (memberof(tp?,gp)).
```

Collision Detection. Collision avoidance is an important property to be checked. In our model, a collision happens when two or more ATVs are located in the same block. We check this property using `search` command which looks for a state with two robots in the same location as follows. The location `(3,16)` denotes the coordinates of the parking station.

```
search [1,1000] initState =>* collisionState such that
              ((locxA == locxB) and (locyA == locyB)) and
              ((locxA =/= 3) and (locyA =/= 16)).
```

Cross Deadlock. Deadlock-freedom is a generic property of a system that must be checked. Specially, we are interested to assure if the defined policies to pass a junction cause no deadlock. The following command checks whether the ATVs `ATVA` and `ATVB` are in deadlock on cross `crossA` or not.

```
red modelCheck(initialmodel,(~<>[]
              oncross(ATVA,crossA)/\oncross(ATVB,crossA))).
```

This property is violated, because when two ATVs with the same priority are at the cross, each ATV waits for the other one to pass. We define a new policy to handle this case: "if the ATVs at a cross have identical priorities, the ATV with the highest ID must pass first“.

The statistics of checking the aforementioned properties are shown in Table 2 in which we list the number of checked states, the number of rewrites, CPU time, maximum reached depth and result (i.e. violated or satisfied). For the verification, we used a PC equipped with an Intel Core 2-Due 2.6 GHz and 6 GByte of memory with Windows 7.

property	**states no.**	**rewrites no.**	**time(sec)**	**depth**	**result**
action conflict	3.9×10^6	1.5×10^{10}	3179	500	S
inexecutable action	4.8×10^5	2×10^9	426	2×10^3	S
collision checking	3.2×10^5	1.3×10^9	2608	2×10^3	V
cross deadlock	1.4×10^6	6×10^9	2976	-	V

Table 2: Verification Results

7. Run-time Verification

Due to the high complexity of an IT Ecosystem model, our model checking analysis encounters the state explosion problem; the size of the state space is far too large to be effectively analyzed. Furthermore, bounded reachability analysis is incomplete. When the configurations of a manager are updated at runtime, we must reverify the new model with the updated managers' configurations. Run-time verification [16, 17] is an attractive complement to design-time verification and a useful technique for verification of HPobSAM models with dynamic configurations. An observer, monitors executions of a software system and verifies that the behavior of the software system adheres to a set of formal specifications. Since only one execution path is examined at a time, the state explosion problem is effectively avoided. Moreover, we can verify the system with dynamic configurations.

7.1. The Approach

Runtime verification can be applied to automatically evaluate the current execution path, either on-line, or off-line by analyzing stored execution traces. We propose an online monitoring approach which incorporates governing policies to monitor the system, in addition to react properly in case of property violation. In our model, an execution trace is a sequence of events emitted by the program. The user provides a specification in the subalgebra CA^a (described in Section 3, presented in [9]) describing how the observed system is expected to behave. This subalgebra is used to specify the actions of governing policies. The models of this specification are all the execution traces that satisfy it. Therefore, we should check finite execution traces generated by the running program against the specification for model conformance, and error messages should be issued in case of failure.

We use a policy-based approach to implement our runtime verification approach and consider an observer to monitor each property. An observer is a manager in our model, who is responsible to (i) monitor the behavior of system through listening to a specific set of events defined as the alphabet of the property, and (ii) check if a received event is expected to occur according to a formal specification of the property formulated as the observer's governing policies. Two classes of simple governing policies are defined for the observer: positive policies which are triggered by *expected events* and negative policies which are activated by unexpected events. An emitted event by the program is expected if it is the head of a trace of the property's models, otherwise this event is unexpected.

A positive governing policy of the observer is used to observe a set of desirable traces expected to happen. The event of such a policy denotes the head of the observing traces, the policy condition indicates the event perquisite which has to be held, and the policy action represents the following traces.

A negative governing policy of the observer is used to detect undesired traces. When an event is received by the observer, it checks if this event is expected to happen; an event is expected to occur if it leads to triggering a positive governing policy, otherwise a violation happens.

An observer is provided with two methods *check* and *violated* which are invoked when respectively, a positive and a negative policy is triggered. The method *check* (See Figure 5) calls a policy synthesis algorithm that translates a property, formulated as a CA^a term, into a set of policies to monitor adherence of the execution path with that property (See Figure 6).

Let ρ indicate a property to be monitored and $L \subseteq \mathcal{E}$ denote its alphabet. We represent ρ in its normal form as follows,

$$\rho \equiv (e_1; \rho_1) + ... + (e_m; \rho_m) \tag{1}$$

where e_i is a conditional term, i.e. $e_i = \phi_i :\rightarrow \varepsilon_i$ and $\varepsilon_i \in L$. A sub-term $(e_i; \rho_i)$, $1 \leq i \leq m$, models a bunch of conditional event traces starting with ε_i when condition ϕ_i is true, and followed by ρ_i. We define a (positive) governing policy $g_i = \langle 2\ , \varepsilon_i\ , \phi_i\ \rangle \bullet \mathit{check}(\rho_i, L, \rho, g_i)$ to monitor the sub-term $(e_i; \rho_i)$. Informally, this governing policy states that the observer should first listen to event ε_i under condition ϕ_i, and afterwards the property ρ_i must be monitored. When event ε_i occurs and policy g_i is triggered, action $\mathit{check}(\rho_i, L, \rho, g_i)$ is executed which leads to replacing policy g_i with a collection of new governing policies to monitor ρ_i.

```
void check(string prop, Arraylist alphabet, string oldProp,string PolicyID)
 {
  // create negative policy set
  if(firstcall)
        policies.add(negPolicySynthesis(alphabet, prop));

  //remove the triggered positive policy
  policies.remove(policyID);

  //retreive the old positive policy set
  oldPosPolicyset = getpolicySet(oldProp);
  if(oldPosPolicyset != null)
    {
       policies.remove(oldPosPolicyset);
       setoldPosPolicyset(oldProp,null);
       }
  //add new positive policies to monitor the following traces
  policies.add(posPolicySynthesis(prop));
   }
```

Figure 5: The method *check* of an observer

Clearly if a positive policy is not activated, it means that the current execution path matches to none of the conditional event traces monitored by that policy. Therefore, all non-triggered policies are removed from the list of the observer's governing policies. In case that none of the positive policies is triggered, it implies that the occurred event matches with no desirable conditional event trace and is unexpected. Consequently, it shows that the property has been violated.

We define a negative governing policy $g'_j = \langle 1 , \varepsilon_j , \top \rangle \bullet violated(\rho)$, for each event $\varepsilon_j \in L$. Since the priority of negative policies is lower than the priority of positive policies, a negative policy is triggered if no positive policy is activated. In other words, when an event matches with no desirable traces monitored by the positive policies, it causes the triggering of a negative policy which means that the event is unexpected.

We illustrate our approach using an example in our scenario. ATVs are not allowed to go to the charging station as long as they are transporting passengers. Hence, we should design the policies of ATVs such that they prohibit robots to go to the charging station while they are transporting passengers. The following property is defined to monitor an ATV behavior where the ATV is in an entrance:

```
 \\ prop is the property to be monitored
Arraylist posPolicySynthesis(string prop)
 {
  if p is empty return null;

  formulate prop as ((c1:->e1); p1) + .... + ((cn:->en); pn);
  for(i=1; i<=n; i++)
   {
     create posPolicy(i)=<2, ei, ci, check(pi)>;
     newpolicies.add(posPolicy(i));
     }
   return newpolicies;
    }

Arraylist negPolicySynthesis(Arraylist alphabet, string prop)
 {
  for(i=1; i<=alphabet.size; i++)
   {
     create negpolicy(i)=<1, ei, true, violate(prop)>;
     negPolicyset.addpolicy(negpolicy(i));
     }
  return negPolicyset;
    }
```

Figure 6: The Policy Synthesis Algorithms

$$\begin{aligned}
A &\equiv (\mathit{energy} < CT :\to (\sum_{x\in\{1,2\}} \mathit{goCStation}(x);\ \mathit{charge}();\ \sum_{z\in\{1,2,3\}} \mathit{goEntr}(z)) \\
&+\ \mathit{energy} \geq CT :\to (\mathit{takePass}();\ \sum_{y\in\{1,2\}} \mathit{goCheckin}(y);\ (\mathit{dropPass}() + \\
&\quad (\mathit{dropPass}();\ \sum_{x\in\{1,2\}} \mathit{goCStation}(x);\ \mathit{charge}()));\ \sum_{z\in\{1,2,3\}} \mathit{goEntr}(z))); A \\
L &= \{\mathit{goCStation}(x), \mathit{charge}(), \mathit{takePass}(), \mathit{dropPass}(), \mathit{goCheckin}(x), \\
&\quad \mathit{goEntr}(x), \mathit{goCStation}(x)\}
\end{aligned}$$

Informally, this property describes two possible sequences of events for the behavior of the ATV. In the first trace, the ATV has limited charge and moves to a charging station. Then, it is charged in the charging station and returns to an entrance. In second case, the ATV has enough energy to take passengers and go to check-in desks 1 and 2. Afterwards, either it moves to the entrances directly or goes to the charging station. The result of the first step of our runtime verification algorithm is as follows:

$$\begin{aligned}
A &\equiv \mathit{energy} < CT :\to \mathit{goCStation}(1); A_1 + \mathit{energy} < CT :\to \mathit{goCStation}(2); A_1 \\
&+\ \mathit{energy} \geq CT :\to \mathit{takePass}(); A_2
\end{aligned}$$

Thus, the initial positive policy set is $g = \{g_1, g_2, g_3\}$ where

$$g_1 = \langle 1, \mathit{goCStation}(1), \mathit{energy} < CT\rangle \bullet \mathit{monitor}(A_1) \quad (2)$$

$$g_2 = \langle 1, \mathit{goCStation}(2), \mathit{energy} < CT\rangle \bullet \mathit{monitor}(A_1) \quad (3)$$

$$g_3 = \langle 1, \mathit{takePass}(), \mathit{energy} \geq CT\rangle \bullet \mathit{monitor}(A_2) \quad (4)$$

Assume the event $takePass()$ occurs and the energy level is greater than CT, then the policy g_3 is triggered. The new positive policy set is $g' = \{g_{31}, g_{32}\}$, obtained by removing g_1 and g_2, and replacing g_3 with the following policies:

$$g_{31} = \langle 1, goCheckin(1), \top \rangle \bullet monitor(A_{21}) \quad (5)$$
$$g_{32} = \langle 1, goCheckin(2), \top \rangle \bullet monitor(A_{21}) \quad (6)$$

It is likely that the ATV energy consumption is underestimated due to unforeseen situation such as longtime waiting in traffic. Hence it may happen that the ATV has to make a plan to go for charging according to its policies, even if it is transporting passengers. In this situation the event $goCStation$ occurs, which leads to activation of none of positive policies. Subsequently, a negative policy is triggered and a violation is reported.

An important and non-trivial issue is how to correct the behavior of a system on-the-fly when properties are violated. The system can be halted to handle this situation, for the runtime verification applied during system simulation. However, for real-time on-line systems, fault diagnosis and system recovery is required, which in general will mean modification of the running system. When such additional capabilities are provided to handle violations, the overall dynamically-monitored system becomes an evolvable system [23]. Thanks to the flexibility of PobSAM, we can modify and/or define policies to react to the violation of properties, without the need to halt the system. Furthermore, as the system evolves by time, the response to a property violation may change. We can simply modify policies used to handle the property violations at runtime. To handle violation in our example, a policy is defined for the ATV to prevent it from going to the charging station, and send a help message to the mobile maintenance vehicle for recharging the ATV.

7.2. Implementation

We use the Java based Ponder2 policy toolkit to develop our transportation scenario. Ponder2 is a self-contained, stand-alone, general-purpose object management system developed at Imperial College [18]. It implements a policy execution framework to develop various applications, and uses message passing between objects. Policies are defined using a high-level language called PonderTalk. Java is used for programming user-extensible managed objects. All elements of a Ponder2 system including domains (groups of managed objects), policies, events and user defined objects, are implemented as managed objects. A publish/subscribe event bus is used for interaction between components and disseminating events which trigger policies [5].

We implement actors as user defined Ponder2 managed objects to which messages are sent for performing an action. Managers of a PobSAM model are implemented using the Ponder2 policy interpreter, and governing and structural adaptation policies are implemented by PonderTalk obligation policies. An obligation policy is specified in the form of an event-condition-action rule, but governing/structural adaptation policies are prioritized event-condition-action rules. Therefore, we convert a governing/structural adaptation policy set into a set of non-prioritized rules [19] to be able to specify them using PonderTalk. We can model sequential composition, non-deterministic choice and conditional choice of policy actions, but it is infeasible to model parallel composition generally. It is worth mentioning that the parallel composition operator is not used to specify the governing and structural adaptation policies in our case study.

Moreover, a behavioral adaptation policy is specified by two obligation policies. An obligation policy, say gp_1, is used for evolving to the adaptation mode and another obligation policy, say gp_2, is defined for switching from adaptation mode to the new configuration. In case that the adaptation mode of a policy is loose adaptation mode, gp_1 is triggered which leads to evolving to loose adaptation mode. When the switching conditions become true, gp_2 is triggered. Then gp_2 deactivates the policies of the old configuration, activates the obligation policies of the new configuration and changes the manager mode to normal. An adaptation policy with strict adaptation mode is modeled similarly with the difference that the policies of the old configuration are deactivated by gp_1 rather than by gp_2. All policies are specified in text files which can be modified at runtime without the need to stop the system. Moreover, we can use an interactive shell provided by Ponder2 to interact with the system, e.g. to add, remove and update policies and agents.

We use an aspect-oriented approach to instrument the code. Aspects are responsible to (i) monitor the behavior of actors, and (ii) send the triggered events to the relevant managers. The managers enforce suitable policies to handle raised events. As an example, an aspect is responsible to check the value of `energylevel` after execution relevant methods. When the `energylevel` becomes lower than a predefined value, event `needcharge` is triggered and sent to the `EnergyManager`.

Flexibility of the Approach. As mentioned before, we are able to modify policies at runtime. Ponder2 provides a shell to communicate with the system, e.g. to modify policies. It is feasible to activate, deactivate and define new policies dynamically. We can deactivate the policy `policy1` using the following command where `root/policy` is the path to access `policy1`:

```
root/policy/policy1 active: false.
```

In the above example, we first define the following policy which prevents the ATV to go to the charging station and calls the maintenance ATV for help:

```
policy := root/factory/ecapolicy create.
          root/policy at:"EnergyManager" put: policy.
          root/policy/EnergyManager event:root/event/urgentCase.
          root/policy/EnergyManager condition:[true].
          root/policy/EnergyManager action:
                      [ :id :mid| root/Airport/TMid stop:id.
                                  root/Airport/TMid callforhelp:mid.].
root/policy/EnergyManager active:true.
```

Then, we create the event `urgentCase` using the following command to trigger this policy where `idX` denotes the ATV's identifier and `midX` denotes the identifier of maintenance ATV:

```
root/event/urgentCase create: #(idX midX).
```

Finally, we deactivate the policy which instructs the ATV to go to the charging station and define new policies for the ATV.

7.3. Evaluation

Analysis. Although, we can verify an abstract model of the system using static analysis, this does not guarantee the correctness of the implementation, because not all informations about an adaptive system are available at design time. For instance, consider to check the property that the ATV should have enough energy to transport the passengers to target X. We have considered the abstract values for the energy level of ATVs in static analysis which makes our analysis inaccurate. For runtime verification, we consider the real values of the energy level and use a more accurate energy consumption function. Hence, we have a more correct and realistic analysis at runtime. It is clear that the problem of state space explosion is avoided at runtime, therefore, we are able to carry out complex analysis.

One of the main analysis which we can perform at runtime is policy analysis. Detection of some policy conflicts could be difficult or even infeasible using static analysis. Thus we can detect these policy conflicts at run-time. Let $g_i = \langle o, e, \phi_i \rangle \bullet a_i$ and $g_j = \langle o, e, \phi_j \rangle \bullet a_j$ denote two arbitrary governing policies. The condition of triggering the policy $g_i(g_j)$ is indicated by $\mathcal{T}_{g_i}$ $(\mathcal{T}_{g_j})$. Let χ_{ρ_i} represent the post-condition of enforcing ρ_i and $e_{\neg\varphi_{g_i}}$ stand for the event where the condition of preserving the effect of g_i's action is falsified. Furthermore, ψ_i represents the prerequisite of executing α_i. Table 3 gives the patterns to detect policy conflicts at runtime. After detecting a policy conflict at runtime, we can use various conflict resolution methods to resolve the policy conflict [24]. In addition, we can check application-specific safety properties such as collision avoidance.

It is worth mentioning that we are only able to check safety properties at runtime, i.e. the properties specifying that something bad should not happen. We can not check liveness properties, i.e. the properties specifying that finally something good will happen. For instance we can not check if a policy is unenforceable, i.e. that policy will never become triggered. This is because of the fact that runtime verification does not consider each possible execution of a system, but just a single or a finite subset. Therefore, it shares similarities with testing: both are usually incomplete.

Performance evaluation. We performed a number of experiments to measure the performance of our approach. We have run the algorithm with different number of ATVs. Figure 7(a) shows the CPU overhead of the algorithm for different number of agents, when we ran the system for 300, 450, 600, 750, 900, 1050 and 1200 seconds. The CPU time overhead of our algorithm varies between 0.5% and 0.75%. Figure 7(b) shows the total number of properties checked for different number of ATVs. Our experiments have been performed on an Intel workstation running Windows 7 with Core 2 Duo 2.2GHz CPU and 4 Gbyte memory.

Conflict Type	**Property**	**Alphabet**
Action Conflict	$\rho_{ac} = \mathcal{T}_{g_i} \wedge \mathcal{T}_{g_j} :\to e_i;\ violated(\rho_{ac}), e_i = e_j$	$L_{\rho_{ac}} = \{e_i\}$
Effect Inference Conflict	$\rho_{eic} = \mathcal{T}_{g_i} :\to e_i;\ \neg\chi_{g_i} :\to e_{\neg\varphi_{g_i}}$	$L_{\rho_{eic}} = \{e_i, e_{\neg\phi_{g_i}}\}$
Inexecutable Action Conflict	$\rho_{iac} = \mathcal{T}_{g_i} :\to e_i;\ \psi_i :\to \alpha_i$	$L_{\rho_{eic}} = \{e_i, \alpha_i\}$

Table 3: Properties for Detection of Policy Conflicts

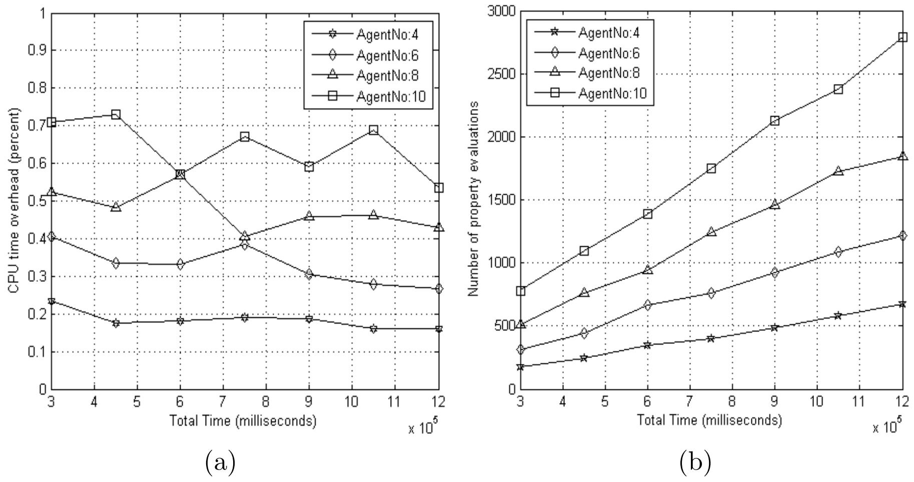

Figure 7: Performance of the Runtime Verification Approach

8. Related Work

Our work is related to different areas of research, however we focus here on related work which intersects ours with modeling adaptive systems, modeling future software systems, formal analysis and runtime verification of self-adaptive systems.

Modeling Self-Adaptive Systems. We have compared PobSAM and HPobSAM with existing formal approaches to model adaptive systems in terms of flexibility, separation of concerns and formal foundation in [9, 13, 14]. Most existing work concentrates on modeling either structural or behavioral changes [6, 11]. The Mechatronic UML is employed for modeling complex mechatronic systems in [27]. It supports the component-based specification of software structure and its adaptation based on graph transformation systems [11]. In this graph transformation-based approach, both behavior and structure are modeled with graphs, however handling large and complex graphs would be difficult for large-scale systems. We take the benefit of both an ordinary state-based formalism for specifying behavioral information in addition to graphs as a natural model to express the system structure. [26] proposed a framework so-called MARS for model-based development of adaptive embedded systems in which a model consists of a set of modules. The MARS model has been extended to optimize the adaptation behavior that emerges hierarchically from modular adaptation behaviors in [38]. This model groups a composition of components together in a hierarchical component, and uses model checking to ensure well-definedness of components. FORM [39] is a formal reference model for self-adaptation based on standardized Z specification language (ISO/IEC 13568:2002) which supports structural adaptation. The scalable models FORM and hierarchal MARS use a bottom-up approach to design the system, however they are inflexible to be used for designing long-lived IT Ecosystems. In contrast to our model, they are only concerned with structural adaptation.

PLASTIC [40] is a formal approach to develop self-adaptive services in which the services should maintain certain levels of quality in changing environments. In this framework, the system is designed by building a model and several variants of the service code are generated from the model to support adaptation at runtime. In contrast to HPobSAM, this application-specific model is neither flexible nor scalable, because this framework only considers design-time adaptations and the system can not evolve at runtime. While PLASTIC uses an automatic approach to generate the code, we manually generate the system code from its HPobSAM model. All in all, the formal approaches for modeling self-adaptive systems are essentially concerned with responding short-term changes (e.g. [6, 11, 25, 26, 27]), while we require scalable approaches which support the long-term evolution of the system. Hierarchical nature and flexibility of PobSAM make it a suitable model to design large-scale systems, as shown in this paper.

Architectures of Self-Adaptive Systems. There are different frameworks and architectures for engineering self-adaptive systems. Rainbow [28] is a framework for self-adaptation which uses an abstract architecture model to monitor the running system. This model is evaluated to control the violation of constraints. In case of violation, several local or global adaptations are performed in the system. In [29], Kramer and Magee propose a three layer architecture for designing self-managed systems: (i) the control component layer implements the functionality of the system, (ii) the change management reacts to the changes at the lower level and executes plans to adapt the behavior of system, and (iii) the goal management layer creates plans to achieve the goals according to the specification of goals and the current state of the system. Similar to [28, 29], in our model the functionality of system is implemented in one layer which is controlled by the upper layers. While Rainbow architecture and Kramer-Magee's reference model are described at an abstract level, we are concerned with the technical details of model and the way that adaptation is performed. Furthermore, our model has a formal foundation compared to [28, 29]. Both Rainbow architecture and Kramer-Magee's model employ a centralized approach to perform adaptation and control the system behavior which is not scalable to design large-scale adaptive systems [21]. We use a combination of centralized and decentralized approaches to design the system. In contrast to [28, 29], our model supports hierarchical structures which enhances the scalability of the model for designing large-scale systems.

Nature-inspired Design of Software Systems. Another relevant area of research is nature-inspired frameworks and models that are proposed to design future software systems. Systems are treated as a spatial Ecosystem in which services, data items, and resources are all modeled as autonomous individuals (agents) that locally act and interact in accordance with a simple set of well-defined eco-laws [1]. There are different nature-inspired metaphors that have been suggested to design future systems including physical metaphors (e.g. [2]), chemical metaphors (e.g. [3]), biological metaphors (e.g. [4]), and social/ecological metaphors (e.g. [30]). According to this definition, policies describe the eco-laws in our model while agents are defined as managers, actors and self-adaptive modules. So far, the proposed approaches have no formal foundation whereas assurance of the correctness of such complex systems is crucial.

Policy-based Design of Software Systems. Furthermore, [5] proposes a policy-based model for engineering ubiquitous computing systems. Our notion of self-adaptive modules is inspired by the self-managed cells in this model. Security and fault management are discussed and the tool-set Ponder2 is provided for implementation. This model has no formal foundation, however event calculus is used to analyze policies. Our model has a formal foundation, we provide an executable formal specification of our model in Maude for analysis purposes, and propose an approach to verify it at runtime. ASSL [31, 32] is a policy-based framework for specification, validation, and code generation of autonomic systems. This approach uses consistency checking to validate the system, and a model-checking approach is provided to verify ASSL models [33]. This flexible three-tier model uses a centralized approach to control the system behavior which makes it non-scalable compared to HPobSAM. While our approach supports both behavioral and structural adaptations, to the best of our knowledge, this model supports only behavioral adaptation.

Formal Verification. Formal verification of adaptive systems at behavioral level is a young research area [34]. Authors in [25] extend LTL with an "adapt" operator called A-LTL to specify adaptation requirements before, during and after adaptation and introduce a model checking approach to verify the program formally. In another work [35], they propose a modular approach to verify adaptive program. [36] present a method to describe adaptation behavior at an abstract level. After deriving transition systems from the system description, the system properties are verified using model checking techniques. In their later work [26], they propose MARS in which the system is specified using Synchronous Adaptive Systems [37] and is verified using theorem proving, model checking and specialized verification methods. All above model checking approaches to verify adaptive systems share the common problem of state explosion with our approach. The modular model checking approaches can enhance the performance of analysis, however, the degree of improvement depends on the granularity of the produced sub-models. It is likely that we encounter the state explosion problem for model checking large sub-model.

Runtime Verification. There are different approaches to verify the adherence of the software behavior to a set of formal specifications. Generally, a desired property to be checked, say ϕ, is expressed and a monitor which accepts all its models is generated. The monitor and the system run in parallel and the system is monitored by the monitor. The language used to specify the property ϕ are usually based on algebra (e.g. [41, 42]), automata (e.g. [43]), logic (e.g. [44, 45]), or regular expressions [46]. In particular, Linear Temporal Logic (LTL) and its different variations have been core to several attempts. Java PathExplorer (JPaX) [47] is a general-purpose monitoring approach to check if the execution of a Java program conforms with a set of user provided properties formulated in temporal logic. JPaX instruments Java byte code to capture the relevant events and send them to the observers. Eagle[48] and RuleR [49] are two rule-based approaches which use a set of rules to monitor the system behavior. In contrast to our approach, the specification language of both approaches is based on temporal logic. Rules in [48, 49] are state-based denoting the state changing in the system while our policies are event-based rules to monitor the system. In [42] a process-algebraic approach is proposed to verify web service compositions at runtime. The specification is transformed into an LTS which is traversed at runtime. While we use a similar language to specify the properties, however, we use a symbolic policy-based approach to implement the monitor. Different from the existing approaches, we used a policy-based approach to implement the monitor. An advantage of a policy-based monitoring approach is that it allows us to deactivate the verification of a property at runtime by disabling the policies used for monitoring that property dynamically. Stopping the verification of a property can be performed due to the reasons such as preventing the performance degradation.

To the best of our knowledge, few researchers concentrate on runtime verification of adaptive systems. In [50], a runtime verification approach is proposed for monitoring adaptive systems. In this work, properties to be checked are specified using LTL and A-LTL. An aspect-oriented approach is used to instrument the code and collect state information for analysis. The model checker AMOEBA [35] is used to analyze the state information, and check the properties. In [51], an LTL-based approach is used to verify properties of service compositions at runtime. Different from [50, 51], we use an algebraic approach to verify properties at runtime. Thanks to flexibility of our policy-based approach, we can also define strategies to handle violation of properties dynamically.

9. Concluding Remarks

In this paper, we studied suitability of HPobSAM for modeling and analyzing IT Ecosystems through a transportation service in a smart airport case study. We discussed different capabilities of HPobSAM including formal foundation, flexibility and scalability. These features make it a suitable model for modeling future software systems. We modeled our case study using HPobSAM, and present formal specification of this model in Maude. We analyzed the case study using the model checking tools provided by Maude at design time. Due to the state explosion problem of model checking, we could not verify all the properties successfully. Moreover, our analysis is done for a model with known managers' configurations, i.e. we have to reverify the system when the managers' configurations are updated at runtime. Hence, we proposed a new run-time verification approach as a complement to model checking which enables us to verify a system with dynamic managers' configurations at runtime. Runtime verification is incomplete compared to (non-bounded) model checking, because we can only check an execution path or a finite subset of execution paths at runtime. Moreover, we have shown how flexibility of our approach can help us to handle property violations at runtime.

References

[1] C. Villalbaa, F. Zambonelli, Towards nature-inspired pervasive service ecosystems: Concepts and simulation experiences, Journal of Network and Computer Applications 34 (2) (2011) 589 - 602.

[2] J. Beal, J. Bachrach, Infrastructure for engineered emergence on sensor/actuator networks, IEEE Intelligent Systems 21 (2) (2006) 10-19.

[3] M. Viroli, M. Casadei, E. Nardini, A. Omicini, Towards a pervasive infrastructure for chemical-inspired self-organising services, Lecture Notes in Computer Science 6090 LNCS (2010) 152 - 176.

[4] W.-M. Shen, P. M. Will, A. Galstyan, C.-M. Chuong, Hormone-inspired self-organization and distributed control of robotic swarms, Autonomous Robots 17 (1) (2004) 93-105.

[5] M. Sloman, E. C. Lupu, Engineering policy-based ubiquitous systems, Computer Journal 53 (7) (2010) 1113-1127.

[6] J. S. Bradbury, J. R. Cordy, J. Dingel, M. Wermelinger, A survey of selfmanagement in dynamic software architecture specifications, In: WOSS, 2004, pp. 28-33.

[7] G. Taentzer, M. Goedicke, T. Meyer, Dynamic change management by distributed graph transformation: Towards configurable distributed systems, In: Proceeding of Theory and Application of Graph Transformations, LNCS 1764, Springer Berlin/Heidelberg, 2000, pp. 179-193.

[8] A. Cansado, C. Canal, G. Salaun, J. Cubo, A formal framework for structural reconfiguration of components under behavioural adaptation, ENTCS 263 (2010) 95-110.

[9] N. Khakpour, S. Jalili, C. L. Talcott, M. Sirjani, M. R. Mousavi, Pobsam: Policy-based managing of actors in self-adaptive systems, ENTCS 263 (2010) 129-143.

[10] J. Zhang, B. H. C. Cheng, Model-based development of dynamically adaptive software, In: Proceedings of the 28th international conference on Software engineering, ICSE 06, ACM, New York, NY, USA, 2006, pp. 371-380.

[11] B. Becker, H. Giese, Modeling of correct self-adaptive systems: a graph transformation system based approach, In: Proceedings of the 5th international conference on Soft computing as trans disciplinary science and technology, 2008, pp. 508-516.

[12] C. Deiters, M. Kster, S. Lange, S. Ltzel, B. Mokbel, C. Mumme, D. Niebuhr, Demsy- a scenario for an integrated demonstrator in a smart city, Tech. rep., NTH - Niedersachsische Technische Hochschule (2010).

[13] N. Khakpour, S. Jalili, C. L. Talcott, M. Sirjani, M. R. Mousavi, Formal modeling of evolving adaptive systems, Science of Computer Programming, 2011, to appear.

[14] N. Khakpour, J. Kleijn, U. Goltz, Integration of structural and behavioral adaptation, submitted.

[15] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, C. L. Talcott, The maude 2.0 system, In: RTA, 2003, pp. 76-87.

[16] I. Lee, S. Kannan, M. Kim, O. Sokolsky, M. Viswanathan, Runtime assurance based on formal specifications, In: PDPTA, 1999, pp. 279-287.

[17] M. S. Feather, S. Fickas, A. V. Lamsweerde, C. Ponsard, Reconciling system requirements and runtime behavior, In: Proceedings of the 9th international workshop on Software specification and design, IWSSD 98, IEEE Computer Society, 1998, pp. 50-.

[18] K. P. Twidle, E. Lupu, N. Dulay, M. Sloman, Ponder2 - a policy environment for autonomous pervasive systems, In: POLICY, 2008, pp. 245-246.

[19] N. Khakpour, R. Khosravi, M. Sirjani, S. Jalili, Formal analysis of policy-based self-adaptive systems, In: Proceedings of ACM Symposium on Applied Computing, Sierre, Switzerland, 2010, pp. 2536-2543.

[20] F. Drewes, B. Hoffmann, D. Plump, Hierarchical graph transformation, J. Comput. Syst. Sci. 64 (2) (2002) 249-283.

[21] H. Schmeck, C. Muller-Schloer, E. Cakar, M. Mnif, U. Richter, Adaptivity and self-organization in organic computing systems, ACM Transaction on Autonomous and Adaptive Systems 5 (2010) 1-32.

[22] D. Weyns, A. Omicini, J. Odell, Environment as a first-class abstraction in multiagent systems, International Journal on Autonomous Agents and Multi-Agent Systems 14 (1) (2007) 5-30.

[23] H. Barringer, D. M. Gabbay, D. E. Rydeheard, From runtime verification to evolvable systems, In: Proceeding of the International Conference Runtime Verification, 2007, pp. 97-110.

[24] N. Dunlop, J. Indulska, K. Raymond, Methods for conflict resolution in policy-based management systems, In: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing, 2003, pp. 98-109.

[25] J. Zhang, B. H. C. Cheng, Specifying adaptation semantics, ACM SIGSOFT Software Engineering Notes 30 (4) (2005) 1-7.

[26] R. Adler, I. Schaefer, T. Schule, E. Vecchie, From model-based design to formal verification of adaptive embedded systems, In: Proceedings of International Conference on Formal Engineering Methods, 2007, pp. 76-95.

[27] S. Burmester, H. Giese, M. Tichy, Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML, In: Proceedings of Model Driven Architecture: Foundations and Applications, LNCS 3599, Springer Verlag, 2005, pp. 47-61.

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

[29] J. Kramer, J. Magee, Self-managed systems: An architectural challenge, In: Proceedings of Future of Software Engineering, FOSE 2007, IEEE Computer Society, Minneapolis, MN, USA, 2007, pp. 259.268.

[30] G. Agha, Computing in pervasive cyberspace, Communication ACM 51 (1) (2008) 68-70.

[31] E. Vassev, M. Hinchey, ASSL: A Software Engineering Approach to Autonomic Computing, IEEE Computer, 42 (6), (2009), 106-109.

[32] E. Vassev, M. Hinchey, The ASSL approach to specifying self-managing embedded systems, Concurrency and Computation: Practice and Experience, (2011).

[33] E. Vassev, M. Hinchey, A. Quigley, Model Checking for Autonomic Systems Specified with ASSL. In: Proceedings of the First NASA Formal Methods Symposium, NFM 2009, NASA, 2009, pp. 16-25.

[34] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, Software engineering for self-adaptive systems: A research roadmap, In: Software Engineering for Self-Adaptive Systems, 2009, pp. 1-26.

[35] J. Zhang, H. Goldsby, B. H. C. Cheng, Modular verification of dynamically adaptive systems, In: Proceedings of Aspect-Oriented Software Development, 2009, pp. 161-172.

[36] K. Schneider, T. Schuele, M. Trapp, Verifying the adaptation behavior of embedded systems, In: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, SEAMS 06, ACM, New York, NY, USA, 2006, pp. 16-22.

[37] I. Schaefer, A. Poetzsch-Heffter, Using abstraction in modular verification of synchronous adaptive systems, In: Trustworthy Software, 2006.

[38] R. Adler, I. Schaefer, M. Trapp, A. Poetzsch-Heffter, Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems, ACM Transactions on Embedded Computer System, 10(2)(2010) 1-39.

[39] D. Weyns, S. Malek, J. Andersson, FORMS: a formal reference model for self-adaptation, In: Proceedings of International Conference on Autonomic Computing, ICAC 2010, Reston, VA, USA, June 7-11, 2010, pp. 205-214.

[40] M. Autili, P. Benedetto, P. Inverardi, Context-Aware Adaptive Services: The PLASTIC Approach, In: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, ETAPS 2009, Springer-Verlag, York, UK, 2009, pp. 124-139.

[41] D. Bartetzko, Jass - Java with Assertions, In: Proceedings of the First Workshop Runtime Verification (RV'01), ENTCS(55)(2), Elsevier Science, Paris, France, Jul. 2001.

[42] M. Khaksar, S. Jalili, N. Khakpour, Monitoring Safety Properties of Composite Web Services at Runtime Using CSP, In : Proceedings of the Middleware for Web Services Workshop at 13th International IEEE EDOC Conference (MWS 2009), IEEE Computer Society Press, Auckland, New Zealand, 2009.

[43] F. Chen and G. Rou, "Toward Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation, Electronic Notes in Theoretical Computer Science, 89(2), 2003.

[44] D. Drusinsky, The Temporal Rover and the ATG Rover, In: Proceedings of SPIN Model Checking and Software Verification, LNCS 1885, Springer, 2000, pp. 323-330.

[45] B. Finkbeiner, H. Sipma, Checking Finite Traces using Alternating Automata, In: Proceedings of the 1st International Workshop on Runtime Verification (RV01), 2001, pp. 44-60.

[46] S. Malakuti, C. Bockisch, M. Aksit, Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software, In: Proceedings of International Symposium on Software Reliability Engineering, 2009, pp. 31-40.

[47] K. Havelund, G. Rou, Monitoring Java Programs with Java PathExplorer, In: Proceedings of the First Workshop Runtime Verification (RV'01), ENTCS(55)(2), Elsevier Science, Paris, France, 2001.

[48] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, In: International 5th Conference on Verification, Model Checking, and Abstract Interpretation, LNCS 2937, Springer, Heidelberg, 2004, pp. 44-57.

[49] H. Barringer, D. Rydeheard, K. Havelund, Rule systems for run-time monitoring: from Eagle to RuleR. In: RV 2007. LNCS 4839, Springer, Heidelberg, 2007, pp. 111-125.

[50] H. Goldsby, B. H. C. Cheng, J. Zhang, "Amoeba-rt: Run-time verification of adaptive software", In: Proceedings of MoDELS Workshops, 2007, pp. 212-224.

[51] J. Camara, C. Canal, G. Salaun, Behavioural self-adaptation of services in ubiquitous computing environments, In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems, 2009, pp. 28-37.