Formal Modeling of Evolving Self-Adaptive Systems

Narges Khakpour[a], Saeed Jalili[a], Carolyn Talcott[b], Marjan Sirjani[c,d], MohammadReza Mousavi[e]

[a] *Tarbiat Modares University, Tehran, Iran*
[b] *SRI International, Menlo Park, California*
[c] *Reykjavk University, Reykjavk, Iceland*
[d] *University of Tehran and IPM, Tehran, Iran*
[e] *Eindhoven University of Technology, Eindhoven, The Netherlands*

Abstract

In this paper, we present a formal model, named PobSAM (Policy-based Self-Adaptive Model), for developing and modeling self-adaptive evolving systems. In this model, policies are used as a mechanism to direct and adapt the behavior of self-adaptive systems. A PobSAM model is a collection of autonomous managers and managed actors. The managed actors are dedicated to the functional behavior while the autonomous managers govern the behavior of managed actors by enforcing suitable policies. A manager has a set of configurations including two types of policies: governing policies and adaptation policies. To adapt system behavior in response to the changes, the managers switch among different configurations. We employ the combination of an algebraic formalism and an actor-based model to specify this model formally. Managed actors are expressed by an actor model. Managers are modeled as meta-actors whose configurations are described using a multi-sorted algebra called CA. We provide an operational semantics for PobSAM using labeled transition systems. Furthermore, we provide behavioral equivalence of different sorts of CA in terms of splitting bisimulation and prioritized splitting bisimulation. Equivalent managers send the same set of messages to the actors. Using our behavioral equivalence theory, we can prove that the overall behavior of the system is preserved by substituting a manager by an equivalent one.

1. Introduction

Motivation Increasingly, software systems are subjected to adaptation at runtime due to changes in the operational environments and user requirements. Adaptation is classified into two broad categories: structural adaptation and

Email addresses: nkhakpour@modares.ac.ir (Narges Khakpour), sjalili@modares.ac.ir (Saeed Jalili), clt@cs.stanford.edu (Carolyn Talcott), msirjani@ut.ac.ir (Marjan Sirjani), m.r.mousavi@tue.nl (MohammadReza Mousavi)

behavioral adaptation [21]. While structural adaptation aims to adapt the behavior by changing the system's architecture, behavioral adaptation focuses on modifying the functionalities of the computational entities.

There are several challenges in designing and developing self-adaptive systems. *Flexibility* is a main concern to achieve adaptation. Due to the fact that today's systems steadily become larger, more heterogeneous, and long-lived, they must have the ability to continuously evolve and grow even in situations unknown during development time. Hence, flexible and scalable approaches are required for developing today's complex and evolving software-intensive systems, since hard-coded mechanisms make tuning and adapting long-run systems complicated. Recently, the use of policies has been given attention as a powerful mechanism to achieve flexibility in adaptive and autonomous systems which allow one to *"dynamically"* specify the requirements in terms of high level goals. A policy is a rule describing under which condition a specified subject must (can or cannot) do an action on a specific object. There are numerous academic and industrial approaches that use policies for managing and adapting the system behavior, e.g., [1, 34] propose architectures for engineering autonomic computing systems, also policies are used as a mechanism to trigger the adaptation in [5, 16, 17, 20, 15].

Due to the fact that self-adaptive systems are often complex systems with a great degree of autonomy, it is more difficult to ensure that a self-adaptive system behaves as intended and avoids undesirable behavior. Hence, one of the main concerns in developing self-adaptive systems is providing mechanisms to ensure that the system is operating correctly, where *model-driven approaches* and *formal methods* can play a key role.

Generally, proposed formal methods for dynamic adaptation mainly focus on the structural changes of adaptive systems (e.g., see [13]). Fewer formal approaches are concerned with the behavioral changes of adaptive systems and developing flexible model-driven approaches to design and develop evolving adaptive systems. Zhang et al. [40] proposed a model-driven approach using Petri Nets for developing adaptive programs. The program consists of a set of steady-state programs and adaptation is done by switching among the steady-state programs. They also presented a model-checking approach for verification of adaptive programs [41, 39] in which an extension of LTL with an "adapt" operator was used to specify the adaptation requirements. Furthermore, authors in [32, 2] proposed a framework named MARS which uses labeled state transition systems directly at a low level of abstraction to model and verify embedded adaptive systems. A MARS module has a set of configurations and adaptation is done by changing the active configuration. All the proposed formal models share the following drawbacks: (1) although, the adaptation concerns are separated from the functionality of the system, the adaptation logic is hard-coded which leads to system's inflexibility. (2) the steady-state programs and configurations are fixed and cannot change dynamically. (3) the system is specified as labeled state transition systems at a low-level of abstraction. Since the adaptation logic as well as configurations/steady-state programs are fixed and cannot change dynamically, these approaches are unsuitable to develop evolving adaptive systems in which the system is required to adapt to unforeseen situations.

This paper In this paper, we propose a formal model called PobSAM (Policy-based Self-Adaptive Model) for developing and specifying self-adaptive systems that employs policies as the principal paradigm to govern and adapt the system behavior. We model a self-adaptive system as a collection of interacting actors directed to achieve particular goals according to predefined policies. A PobSAM model is composed of a collection of autonomous managers and managed actors. Autonomous managers are meta-actors responsible for monitoring and handling events by enforcing suitable policies. Each manager has a set of configurations containing adaptation policies and governing policies. A manager changes its configuration dynamically in response to the changing circumstances according to adaptation policies. The behavior of managed actors is either governed by managers or cannot be directly controlled from outside. Governing policies are the rules that are applied while the system is in a stable state. Adaptation policies are rules that govern the transient states between two stable states while the system is changing. The set of manager's configurations is not fixed and may change dynamically by growing and evolving the system.

PobSAM has a formal foundation that employs an integration of algebraic formalisms and Actor-based models. The computational (functional) model of PobSAM is actor-based while the multi-sorted algebra CA (Configuration Algebra) is proposed to specify the configurations of managers. The sub-algebra CA^a is used to specify actions of simple governing policies which is built on the process algebra PA$_\delta^{cc}$ [10]. The common set theoretic operators are employed to specify governing policy sets. We formalize adaptation policies using CA^p. The operational semantics of PobSAM is described with labeled transition systems.

We provide a behavioral equivalence to reason about PobSAM models. A sound and complete axiomatizations, modulo splitting bisimulation, is provided for CA^a. We introduce an operator, named Ψ, to formulate the behavior of governing policy sets in terms of CA^a terms. Then we use CA^a axiom system to reason about the behavioral equivalence of governing policy sets. We introduce a new type of bisimilarity named *prioritized splitting bisimulation*. A sound and complete axiomatizations, modulo prioritized splitting bisimulation is proposed to describe the behavioral equivalence of adaptation policies. Following [11], we define a particular model for CA^p in terms of prioritized conditional transition systems modulo prioritized splitting bisimulation, which indeed satisfies the defined axioms of CA^p. Furthermore, we present an equational theory to reason about configurations and managers. This equational theory is particularly helpful for automatic component assessment. We refer to component assessment as the problem of identifying a component with desired behavior that can replace another component or can be used for interaction. A possible solution to this problem relies on detecting the behavioral equivalence of a particular component with desired behavior and a candidate component that could maintain that behavior. We will illustrate this problem by means of concrete examples.

Contributions In the past, rigid formal methods have been proposed for modeling and analysis of adaptive systems at the behavioral level, mainly at low levels of abstraction, and flexible policy-based approaches have been proposed for designing adaptive systems without formal foundation. Here, we propose a flexible policy-based approach with formal foundation to support model-driven developing and verification of self-adaptive systems. Compared to existing work, our approach has the following novel features:

1. PobSAM is a *novel flexible formal approach* to design and develop evolving self-adaptive systems which uses an identical mechanism, i.e., policies, to adapt and control the system behavior.
2. Policies allow us to separate the rules that govern the behavioral choices of a system from the system functionality, giving us a higher level of abstraction, so that we can change the system behavior by enforcing suitable policies which can be modified at runtime, without the need to change the system code. As an example, we are able to change and reason about the scheduling of jobs using policies independent of the system code.
3. Policies are specified at a high-level of abstraction and allow us to decouple the adaptation concerns from the application code. We can adapt the system behavior to unforeseen situations by defining and modifying the policies dynamically (i.e., governing policies and adaptation policies). This is a major advantage over the proposed formal models for developing evolving and complex systems.
4. Safe adaptation is a main requirement of a model to develop self-adaptive systems. To this end, we present an adaptation strategy to pass the adaptation phase safely and at the right moment.
5. Since, PobSAM is a modular model, it can support both structural and behavioral adaptation, however in this paper, we focus on the behavioral adaptation.
6. The formal foundation, the modular model, and separation of adaptation rules will help us in developing rigorous analysis techniques. Since, CA^a is complete and sound for splitting bisimulation and CA^p is complete and sound for prioritized splitting bisimulation, we can reason about policy actions, governing policies, adaptation policies, configurations and managers separately, without need to construct the whole model of the system. This is a main advantage for evolving systems whose requirements and environment change dynamically.

Compared to the conference version of this paper [22], in this paper (1) we axiomatize CA to specify manager configurations (2) we present a behavioral equivalence theory to reason about CA terms, and (3) we apply our approach on a more concrete case study.

Structure of the paper This paper is organized as follows. In Section 2 we introduce an example to illustrate our approach. Section 3 briefly introduces the PobSAM model. Sections 4 and 5 introduce the syntax and the semantics of PobSAM model, respectively. A notion of behavioral equivalence for CA is presented in Section 6. Section 7 presents related work and compares our approach with the existing approaches. In Section 8, we present our conclusions and plans for future work.

2. Illustrating Case Study

We use a simple example borrowed from [34] to illustrate our approach. In this example, a team of collaborating unmanned autonomous vehicles (UAVs) are used for a search and rescue operation. Each UAV is provided with a video camera, GSM for location sensing, and infrared sensors for detecting physical obstacles. Moreover, a UAV has the basic capability to control its movement. Different technologies are used for UAVs communications, including WiFi for interacting with other vehicles, and satellite or cellular 3G (in urban environments) for long distance interactions.

Assume a person with a body sensor network (BSN) is wounded in an area and needs help. A set of UAVs with different capabilities collaborate with each other to find the wounded person. There is an autonomous device named *mission commander* to coordinate the rescue and save operation. The BSN sends a help message to a mission commander. A mission is defined by the commander to save the wounded person: one or more UAVs with video cameras act as surveyors and others perform a communication relay function.

The UAVs are required to adapt their behavior according to the changes in environment. The role of a UAV is not fixed, and it can be assigned to different roles according to its available capabilities. For instance, the video camera of a surveyor may break down and the surveyor would be assigned as a communication relay. Moreover, the responsibilities of a role may change dynamically to adapt to unforeseen situations. Suppose we encounter this situation after setting up the mission: the mentioned area may have hazardous chemicals. Thus we should use UAVs equipped with some sensors for detecting chemicals to locate the wounded person. Hence, we should design the system in a flexible manner which allows us to change the behavior of a UAV dynamically. We will show that how we address these requirements in the examples 5 and 6.

Another reason of adaptation in our scenario is that the UAVs must cope with variable resources and faults. To this end, we require to find suitable UAVs for interaction or replacement, e.g., consider a situation that a surveyor breaks down and requires to be replaced by another UAV with the identical surveying capabilities. Thus, we need to identify the UAVs with the equivalent behavior at runtime that can replace the surveyor. We will deal with this issue in the examples 12 and 13.

3. The Outline of PobSAM

As mentioned above, PobSAM is a policy-based formal model to develop and specify self-adaptive evolving systems. The main elements of a PobSAM model are actors and managers: actors perform the main functionality of the system, and managers control the behavior of actors autonomously according to a set of predefined policies. Furthermore, views are abstractions of the actors provided for the managers. The PobSAM structure can be conceptualized as the composition of three layers:

- **Managed Actors Layer** This layer is dedicated to the functional behavior of a system and contains computational actors. Actors are governed by autonomous managers using policies to achieve predefined goals. Henceforth, we use the terms managed actors and actors interchangeably.

- **Autonomous Managers Layer** Autonomous managers are meta-actors that can operate in different configurations. Each configuration consists of two classes of policies: governing policies, and adaptation policies. Using governing policies, the manager directs the behavior of actors by sending messages to them. Adaptation policies are used to switch between different configurations to adapt the system behavior properly.

- **View Layer** In PobSAM, each actor provides its required state information to the relevant managers. Not all aspects of the operational environment have direct influence on the behavior of managers. The view layer is composed of a set of *view variables*, and provides an abstraction of the actor states that is adequate for the managers' needs. The distinction between the underlying computational environment and the required state information of actors makes analyzing managers much simpler.

Example 1. The PobSAM model of a UAV contains the actors *motor*, *video camera*, *GSM* and *infrared sensors*. Although, in general, actors may have interaction with each other, in this example, there is no interaction among the actors. The view layer has a number of view variables indicating attributes such as the current location and speed. A UAV has a manager, named *UAVCntrlr*, which controls different components of the UAV. We consider a configuration for each role of UAV where one of the configurations is activated at each time. The UAV can change its role by switching to the suitable configuration.

4. PobSAM Syntax

A PobSAM model is denoted by $\Pi = \langle R, V, E, M \rangle$ in which R, V, E and M represent the set of actors, view variables, events and managers, respectively. Figures 1 and 2 show the syntax and a BNF grammar of PobSAM model, respectively, which we elaborate in the sequel.

4.1. Actors

The encapsulation of state and computation, and asynchronous communication make actors a natural way to model distributed systems. Therefore, we use an actor-based model to specify the computational environment of a self-adaptive system. To this end, an extension of Rebeca [33] is used. Rebeca is an actor-based language for modeling concurrent asynchronous systems which allows us to model the system as a set of reactive objects called rebecs interacting by message passing. Each rebec provides methods called message servers (*msgsrv*) which can be invoked by others. Each rebec has an unbounded buffer for incoming messages, called queue. Furthermore, the rebecs' state variables

```
MANAGERS {
  MANAGER ManagerName1
    // manager's view
    view = { viewvarName11,..., viewvarName1n};
    // definition of manager's configurations in terms of
    // their governing and adaptation policies
    Configurations {
        configName1={gp11,...,gp1m}{ap11,...,ap1n};
        .
        .
        .
    }
    Policies {
        //Definition of governing policies(gps)
        gp11: on eventi if condi do actionsi priority Oi;
        .
        .
        .
        //Definition of adaptation policies(aps)
        ap11:[loose/strict]
          on eventj if condj switchto Configuration1 when condk priority Oj;
        .
        .
        .
     }
   }
  // definition of other managers
}
Viewvariables {
  //definition of views
  datatype viewvarName1 = expr1;
  datatype viewvarName2 = expr2;
    .
    .
    .
}
Events {
  //definition of events
  eventName1 = expr1;
  eventName2 = expr2;
    .
    .
    .
}
Actors {
  //definition of actors
  reactiveclass classname1() {
     Knownrebecs{}
     Statevars{  Public datatype v1;
                 Private datatype v2;}
     msgsrv initial() {}
     msgsrv msgsrv1(){}
  }
}
  main {
     classname1 rebec1(...):(...);
     // instantiate other rebecs
     ManagerName1 manager1(viewNamei,viewNamej)(configNamek);

  }
```

Figure 1: Syntax of PobSAM Models

$$\begin{array}{l}
CL ::= \textbf{reactiveclass}\ CId(Nat)\ \{KRs\ Vars\ Mtd^{*}\} \\
KRs ::= \textbf{knownrebecs}\{\langle Vdcl;\rangle^{*}\} \\
Vars ::= \textbf{statevars}\{\langle Vdcl;\rangle^{*}\} \\
Vdcl ::= T\langle v\rangle^{+}_{,} \\
Mtd ::= \textbf{msgsrv}\ M(\langle T\ v\rangle^{*}_{,})\{St^{*}\} \\
Call ::= v.M | \textbf{self}.M | \textbf{sender}.M
\end{array}$$

$$\begin{array}{l}
St ::= v = exp; \\
\quad |v = \textbf{new}CId(\langle exp\rangle^{*}); \\
\quad |Call(\langle exp\rangle^{*}_{,}); \\
|\textbf{if}\ (exp)\ \langle St^{*}\rangle\ \langle \textbf{else}\ St^{*}\rangle^{?}
\end{array}$$

$$\begin{array}{l}
Mng ::= \textbf{manager}\ m\ \{Cfs\ Pls\ Vws\} \\
Cfs ::= \textbf{configurations}\ \{\langle Cfdcl\rangle^{+}\} \\
Cfdcl ::= cf = \{\langle gp\rangle^{*}_{,}\}\{\langle ap\rangle^{*}_{,}\} \\
Pls ::= \textbf{policies}\ \{\langle Pldcl;\rangle^{*}\} \\
Pldcl ::= \langle Gpdcl \mid Apdcl\rangle \\
Apdcl ::= ap : [\textbf{loose}|\textbf{strict}]\ \textbf{on}\ e\ \textbf{if}\ \ exp\ \ \textbf{switchto} \\
\qquad\qquad\qquad\qquad cf\ \textbf{when}\ \ exp\ \ \textbf{priority}\ Nat; \\
Gpdcl ::= gp : \textbf{on}\ e\ \textbf{if}\ \ exp\ \ \textbf{do}\ Acdcl\ \textbf{priority}\ Nat; \\
Acdcl ::= (v.M \mid exp :\rightarrow Acdcl)((+ \mid \| \mid ;)(v.M \mid exp :\rightarrow Acdcl))^{*} \\
Vws ::= \textbf{views} = \{\langle vw\rangle^{*}_{,}\}; \\
Views ::= \textbf{Viewvariables}\ \{\langle T\ vw = exp\ ;\rangle^{*}\} \\
Events ::= \textbf{Events}\ \{\langle e = exp\ ;\rangle^{*}\}
\end{array}$$

Figure 2: BNF grammar for Rebeca classes, managers, views and events. Angle brackets $\langle ::: \rangle$ are used as meta parentheses, superscript ? for optional parts, superscript + for repetition more than once, superscript * for repetition zero or more times, whereas using $\langle ::: \rangle_{,}$ with repetition denotes a comma separated list. Identifiers *C, T, M* and *v* denote class, type, method and variable names, respectively; Nat denotes a natural number; and, exp denotes an (arithmetic, boolean) expression. *m, cf, gp, ap, e* and *vw* indicate the identifiers of manager, configuration, simple governing policy, adaptation policy, event and view, respectively.

(*statevars*) are responsible for capturing the rebec state. The known rebecs of a rebec (*knownrebecs*) denotes the rebecs to which it can send messages. In our simple extension, an actor can expose a number of its state variables to the managers, i.e., an access specifier (private or public) is defined for state variables (Figure 1, block `Actors`). The public state variables are used in the definition of view variables, and private state variables are accessed only by the owner rebec.

Example 2. Figure 3 shows the actor layer of a UAV partially. We consider a reactive class named `motor` to model the motors which contains `forward`, `backward`, `stop` and `setSpeed` message servers, as well as `motorPort` and `speed` state variables. `UAV1motor` and `UAV2motor` are rebecs instantiated from `motor` which model the motors of `UAV1` and `UAV2`, respectively.

4.2. Views and Events

In PobSAM, the view layer is defined as a set of view variables. A view variable is a function defined over public state variables of actors, i.e., $v = f(x_1, ..., x_n)$, $v \in V$, where $x_1, .., x_n$ indicate the public state variables of

```
Viewvariables {
  byte speed1 = UAV1motor.speed;
  byte location1 = GSM1.location;
  byte location2 = GSM2.location;
  //definition of other view variables
}
Actors{
 reactiveclass motor() {
  knownrebecs {}
  statevars{ public byte motorPort;
            public byte speed; }
  msgsrv initial() { //initialization }
  msgsrv forward()  {
    ...
   }
  msgsrv backward()  {
    ...
   }
  msgsrv stop()  {
    ...
   }
  msgsrv setSpeed(byte s)  {
    ...
  }
 }
 reactiveclass GSM() {
    ...
   }
} //definition of other reactive classes
main {
 motor UAV1motor():();
 motor UAV2motor():();
 //instantiation of other rebecs
 ...
 }
```

Figure 3: The Actors and View Layers of the UAV example

actors. Unlike conventional interfaces, a view variable can be defined over state variables of different actors. View variables enable managers not to be concerned about the internal behavior of actors and they provide an abstraction of actor's state to managers. Figure 3 gives the view layer of the UAV example partially.

Events are defined using the following predefined predicates:

- *removed*(x) when an actor with specification x is removed,
- *created*(rc) when an actor is instantiated from the reactive class rc,
- *sentMsg*(src, msg, trg) when a message msg is sent from src to trg,
- *exeMsgsrv*(r, msg) when the execution of message server msg is completed by actor r, and
- *prd* when a specific condition in the system becomes true.

4.3. Managers

Managers direct and adapt the behavior of actors by enforcing suitable policies and the view layer provides contextual information for the managers. A manager may have different configurations, of which one is active at each time, and dynamic adaptation is performed by switching among them. A manager has access to a set of view variables of the view layer (see Fig.1). Let $V = \{v_1, ..., v_n\}$ denote the view layer of model. Formally, manager such as m is defined as the tuple $m = \langle V_m, C, c_{init} \rangle$ where $V_m \subseteq V$ indicate the view of m, $C = \{c_1, ..., c_n\}$ denotes the set of m's configurations, and c_{init} is the initial configuration of m. A configuration c_i is defined as $c_i = \langle g, p \rangle, 1 \leq i \leq n$ where g and p are the set of governing policies and adaptation policies of c_i, respectively. A policy is a rule which is triggered when a specific event occurs and some conditions hold.
Example 3. Assume `UAV1` can operate as a *surveyor* or a *relay*, or stays idle. This robot starts the mission as a *surveyor*. The `UAV1`'s controller is defined as follows:

$UAVCntrlr1 = \langle V_m, \{surveyor, relay, idle\}, surveyor \rangle$,

where $\{speed1, location1, location2\} \subset V_m$. We consider $location2$ as a view variable of `UAV1` because it needs to know the location of `UAV2` for collision avoidance.

We define configurations formally using an algebraic theory called CA (for <u>C</u>onfiguration <u>A</u>lgebra). The algebraic theory CA is a multi-sorted algebra containing four sorts: the sort **G** of governing policy sets, the sort **A** of governing policy actions, the sort **P** of adaptation policies, and the sort **B** of conditions. CA consists of subtheories CA^a, CA^p and $\mathcal{B}$ to define terms of sorts **A**, **P** and **B**, respectively. Moreover, CA^g is a common set algebra to define governing policy sets. A fixed but arbitrary set of atomic conditions B_{at} are assumed whose atomic predicates are defined over the view variables. The constants $\top \in B_{at}$ and $\bot \in B_{at}$ stand for "True" and "False", respectively. $\mathcal{B}$ is the Boolean algebra over B_{at} and conditions of CA are $\mathcal{B}$ terms.

Governing Policies

The governing policy set g of a configuration is a collection of simple governing policies, i.e., $g = \{g_1, ..., g_n\}, n \geq 0$. A simple governing policy is defined as a prioritized event-condition-action rule. Whenever a manager receives an event, it identifies all the simple governing policies that are triggered by that event. For each of these policies, the policy condition is evaluated. If the condition evaluates to true, the action part of the triggered policy is requested to execute by sending asynchronous messages to the relevant actors. If multiple simple governing policies become activated, they are enforced sequentially in any arbitrary order.

A simple governing policy $g_i = \langle o,e,\psi \rangle \bullet a$, $g_i \in g, 1 \leq i \leq n$, consists of priority $o \in \mathbb{N}$, event $e \in E$, condition ψ and action a. ψ is a term of sort **B** and a is non-recursive term of sort **A**. Furthermore, $\mathbb{N}$ is the set of natural numbers.

The action part of a simple governing policy is specified using CA^a which is a sub-theory of the process algebra PA$_\delta^{cc}$ [10]. The encapsulation and pre-abstraction operators of PA$_\delta^{cc}$ are excluded from our algebra. PA$_\delta^{cc}$ is a simple process algebra with a strong theoretical foundation which supports conditional expressions. It is assumed that a fixed but arbitrary finite set of primitive actions $\mathcal{A}$ with typical elements $\alpha, \beta, ...$ has been given. Composite actions are constructed from primitive actions using CA^a's operators. Henceforth, let A represent the closed terms of sort $\mathbf{A}$, $\{\phi,\ \psi\} \in B_{at}$ and $\{a, a', a''\} \in A$. The algebraic theory CA^a has the following constant and operators to build terms of sort $\mathbf{A}$:

$$a \stackrel{\text{def}}{=} a;a' \ \mid\ a \parallel a' \ \mid\ a \mathbin{\|\!\lfloor} a' \ \mid\ a + a' \ \mid\ \phi :\rightarrow a \ \mid\ \alpha \ \mid\ \delta_a$$

Thus an action term can be the sequential (;) or parallel execution ($\parallel$) of actions. The operator $\mathbin{\|\!\lfloor}$ is as $\parallel$ but the first action that is performed comes from the left operand. This is an auxiliary operator required to axiomatize parallel composition. The term ($\phi :\rightarrow a$) represents that action term a can be chosen to be executed, if ϕ holds. Also, the manager can execute the actions non-deterministically (+), of which the arguments may be guarded by a condition. Moreover, the special constant δ_a is the deadlock and can perform no activity and prevents subsequent processes from being executed. The primitive actions of the action of a simple governing policy are of the form $r.\ell(v_1, ..., v_n)$ standing for the sending of message $\ell(v_1, ..., v_n)$ to the actor r.
Example 4. The simple governing policy "Get health information of the wounded person from his BSN and send a "success" message to the commander" in the *surveyor* configuration is specified as follows where `found(x,y)` denotes the event that the wounded person has been found at location (x, y). Furthermore, `send` and `Healthinfo` are the message servers of the actors `relay1` and `BSN`, respectively.

```
g1: on found(x,y) if true do
           BSN.getHealthInfo()|| relay1.send([success,x,y], commander)
        with priority 1
```

The algebraic form of this policy is $g_1 \stackrel{\text{def}}{=} \langle 1,\ found(x, y),\ \top\rangle \bullet\ a_1$ where

$$a_1 \stackrel{def}{=} \mathit{BSN.getHealthInfo}() \ \parallel\ \mathit{relay1.send}([\mathit{success}, x, y], \mathit{commander})$$

Adaptation Policies
One of the main features of a formal model which specifies a self-adaptive system is the adaptation semantics. To this end, we should deal with a number of issues such as "when an adaptation is performed in the system " , "when the manager's policies are modified" , "when the enforcement of new policies begins after modifying policies" or "how to restrict the system behavior during adaptation".

Whenever an event requiring adaptation occurs, relevant managers are informed. However, adaptation cannot be done immediately and only when the system reaches a safe state, the concerned managers switch to the new configuration. Therefore, we introduce a new mode of operation named adaptation mode in which a manager runs before switching to the next configuration. While the manager is in the adaptation mode, it is likely that events occur which need to be handled by managers. To handle these cases, we introduce two kinds of adaptations named *loose adaptation* and *strict adaptation.* Under loose adaptation a manager enforces old policies, while under strict adaptation all events will be postponed until the manager exits the adaptation mode and the system reaches a safe state. As an example, consider a situation in which an adaptation is required to use a relay as a surveyor. To this end, we must replace that UAV's governing policies with the new policies for surveying the area. The relay must deliver its current messages to the receivers firstly, and afterward it should act as a surveyor. Thus, the UAV must go to the *loose adaptation* mode according to the introduced adaptation semantics of PobSAM. The behavior of UAV is restricted in the loose adaptation mode: it is prohibited to receive new messages from other UAVs while the old messages are being delivered.

A simple adaptation policy is a prioritized rule that whenever triggered, the manager evolves to adaptation mode and waits until the system reaches a safe state. The manager will switch to the new configuration when a safe state is reached. Adaptation policies are specified using the sub-algebra CA^p which constructs terms of sort **P** as follows:

$$p \stackrel{\text{def}}{=} \langle o, e, \psi, \lambda, \phi \rangle \bullet c \mid p \oplus p \mid \delta_p$$

in which $o \in \mathbb{N}$ denotes the priority of adaptation policy, $e \in E$ indicates an event, and ψ denotes the condition of triggering adaptation. Moreover, ϕ and λ indicate the conditions of applying adaptation and the adaptation type (loose or strict) while c is the new configuration. ψ, λ and ϕ are terms of sort **B**. Values $\top$ and $\bot$ of λ denote strict and loose adaptations, respectively. Informally, the simple adaptation policy $\langle o, e, \psi, \lambda, \phi \rangle \bullet c$ means when event e occurs and the triggering condition ψ holds, if there is no other triggered adaptation policy with a priority higher than o, the manager evolves to the strict or loose adaptation modes based on the value of λ. When the condition of applying adaptation ϕ becomes true, it will perform adaptation and switch to the configuration c. Adaptation policies of a manager are defined as composition ($\oplus$) of the simple adaptation policies. Furthermore, δ_p indicates the unit element for the composition operator.

Example 5. Assume a situation in our example that the video camera of surveyor breaks down. This UAV is used as a relay in the mission, and another UAV with a camera can play the surveyor role. We define an adaptation policy which states "when the camera of a surveyor breaks down, if the wounded person has not been found yet and the surveyor has enough energy, it should switch to the *relay* configuration". We specify this policy formally as formula 1 in which `brokencamera` is an event. The view variable `enoughEnergy` indicates if the energy level of the UAV is sufficient, and the view variable `success` denotes whether the wounded person has been found or not.

```
p1: [loose] on brokencamera if (!success && enoughEnergy) switchto relayconf
```

```
                when true with priority 1;
```

The algebraic form of this policy is as follows:

$$p_1 \overset{def}{=} \langle 1, brokencamera, \neg success \wedge enoughEnergy, \bot, \top \rangle \bullet relayconf \qquad (1)$$

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 at runtime which leads to changing the behavior of system consequently. Thus, PobSAM allows us to adapt to unforeseen situations without need to modify the system code by simply defining new set of policies. When the manager receives a message $add(c)$, it will add configuration c to the configuration list of the manager, if c is not in the configuration list of the manager. In case that the manager receives a message $remove(c)$, it will remove c from the configuration list of the manager provided that c is not the current configuration of the manager and belongs to the configuration list of the manager. Furthermore, when the manager receives a message $load(c, \lambda, \phi)$, if c is not the current configuration of the manager, it evolves to the strict or loose adaptation modes based on the value of λ. When the condition of applying adaptation ϕ becomes true, it will switch to the configuration c.

Example 6. Assume a situation that the area is contaminated with hazardous chemicals. We must be able to change the mission to adapt to these circumstances. A couple of UAVs must be responsible to detect chemicals and locate the wounded person. We simply define a new configuration using a high-level language containing suitable policies to detect chemicals, and load this configuration to the UAVs equipped with hazard detector sensors. Afterward, the relevant UAVs are instructed to switch to this new configuration and use new loaded policies to search the area.

5. Operational Semantics of PobSAM

In this section, we present the operational semantics of PobSAM. First, we present the operational semantics of the view layer, then we explain the structural operational semantics of the main part of PobSAM, i.e., managers. The operational semantics of our simple extension of Rebeca with access specifiers does not differ from that of Rebeca [33].

5.1. Operational Semantics of the View Layer

Any changes in the actors' states used in the definition of view variables, must be reflected in the view layer. The state of a view variable is determined by its current value that is modified by changing the relevant public state variables of actors. After execution of a message server, the changes of public state variables must be reflected in the view variables state, too. We specify the operational semantics of the view layer as a labeled transition system defined based on the semantics of the actor layer. The operational semantics of the actor layer is defined as the labeled state transition system $T_A = (s_a^0, S_A, L_A, T_A)$ where s_a^0 indicates the initial state of the actor layer, S_A is the set of actor states, L_A is the set of labels, and $T_A \subseteq S_A \times L_A \times S_A$ represents the transition relation.

Let $V = \{v_1, v_2, \ldots, v_n\}$ denote the view layer of the model, and $v_j = f_j(x_1, x_2, .., x_m)$ denote an arbitrary view variable defined on public state variables $x_1, x_2, .., x_m$. The operational semantics of the view layer is defined as $T_V = (s_v^0, S_V, L_V, T_V)$ where where s_v^0 indicates the initial state of the view layer, S_V is the state set of view layer, $L_V \subseteq L_A$ is the set of labels, and $T_V \subseteq S_V \times L_V \times S_V$ represents the transition relation. The states of S_V are of the form $s_v = \langle v_1, v_2, ..., v_n \rangle$. Let $v_j|_{s_a}$ denote the value of v_j in which $x_i, 1 \leq i \leq m$ is substituted with its corresponding value in state $s_a \in S_A$. The initial state of the view layer is defined as $s_v^0 = \langle v_1|_{s_a^0}, ..., v_n|_{s_a^0} \rangle$. The state transition relation of the view layer is built based on the state transition relation of the actor layer using rule VR. This rule states when the actor layer evolves from state s_a to state s'_a and there exists a view variable such as v_k whose value changes by this transition, the view layer switches to a new state to reflect the changes of the actor layer.

$$(\mathrm{VR})\frac{s_a \xrightarrow{l} s'_a \qquad \exists_k \; v_k|_{s_a} \neq v_k|_{s'_a}, 1 \leq k \leq n}{\langle v_1|_{s_a}, ..., v_n|_{s_a} \rangle \xrightarrow{l} \langle v_1|_{s'_a}, v_2|_{s'_a}, ..., v_n|_{s'_a} \rangle}$$

5.2. Operational Semantics of Managers

We use prioritized conditional state transition systems to define the operational semantics of CA. Prioritized conditional state transition systems are an extension of conditional state transition systems [10] with priorities defined as follows:

Definition 1. Prioritized Conditional State Transition Systems A prioritized conditional state transition system is defined as $\mathrm{T} = \langle S, \rightarrow, \rightarrow_{\surd}, s_0 \rangle$ where S is a set of states, $s_0 \in S$ is the initial state, and for each $l \in \mathcal{B}^- \times \mathcal{A} \times \mathbb{N}$, $\xrightarrow{l} \subseteq S \times S$, $\xrightarrow{l}_{\surd} \subseteq S$ and $\mathcal{B}^- = \mathcal{B} \backslash \bot$ is the set of Boolean terms excluding $\bot$.

For convenience, we write $s \xrightarrow{(\phi,\alpha,n)} s'$ instead of $(s, s') \in \xrightarrow{(\phi,\alpha,n)}$ and $s \xrightarrow{(\phi,\alpha,n)} \surd$ instead of $s \in \xrightarrow{(\phi,\alpha,n)} \surd$. $s \xrightarrow{(\phi,\alpha,n)} s'$ means that it is possible to perform action α under condition ϕ in state s when there is no enabled transition with higher priority than n in state s, and then make a transition to s'. $s \xrightarrow{(\phi,\alpha,n)} \surd$ is interpreted as in state s, it is possible to perform action α under conditions ϕ when there is no enabled transition with higher priority in state s and then terminate successfully. Henceforth, we denote a transition by $s \xrightarrow{\mu} s'$ where $\mu = (\phi, \alpha, o)$.

A manager has four running modes, including waiting, loose adaptation, strict adaptation and governing policy enforcement modes. The behavior of a

$$(\text{NPE1})\frac{\text{Trig}(e,v)=\{g_i{\in}g \mid v{\vDash}\psi_i,e_i{=}e,\nexists g_j{\in}g.v{\vDash}\psi_j{\wedge}e_j{=}e{\wedge}o_j{>}o_i\} \qquad \text{Trig}(e,v){\neq}\emptyset}{\mathcal{M}_C^c\langle\delta_p,\emptyset,\surd,observe(e){:}q\rangle\xrightarrow{(\mathcal{T}_{g_i}(e),observe(e),1)}[\mathcal{M}]_C^c\langle\delta_p,\text{Trig}(e,v),\surd,q\rangle}$$

$$(\text{NPE2})\frac{g_i{\in}g'}{[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle\xrightarrow{(\top,enforce(g_i),1)}[\mathcal{M}]_C^c\langle\delta_p,g'\backslash g_i,\ g_i.a,q\rangle}$$

$$(\text{NPE3})\frac{}{[\mathcal{M}]_C^c\langle\delta_p,g',\alpha,q\rangle\xrightarrow{(\top,\alpha,1)}[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle}$$

$$(\text{NPE4})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a'',q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',a{+}a',q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a'',q\rangle} \quad (\text{NPE5})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',a{+}a',q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle}$$

$$(\text{NPE6})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{(\phi,\alpha,1)}[\mathcal{M}]_C^c\langle\delta_p,g',a',q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',\psi{:}{\rightarrow}a,q\rangle\xrightarrow{(\psi\wedge\phi,\alpha,1)}[\mathcal{M}]_C^c\langle\delta_p,g',a',q\rangle}$$

$$(\text{NPE7})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{(\phi,\alpha,1)}[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',\psi{:}{\rightarrow}a,q\rangle\xrightarrow{(\psi\wedge\phi,\alpha,1)}[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle}$$

$$(\text{NPE8})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a'',q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',a\|a',q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a''\|a',q\rangle}(\text{NPE9})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',a\|a',q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a',q\rangle}$$

$$(\text{NPE10})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a'',q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',a;a',q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a'';a',q\rangle}(\text{NPE11})\frac{[\mathcal{M}]_C^c\langle\delta_p,g',a,q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',\surd,q\rangle}{[\mathcal{M}]_C^c\langle\delta_p,g',a;a',q\rangle\xrightarrow{\mu}[\mathcal{M}]_C^c\langle\delta_p,g',a',q\rangle}$$

$$(\text{NPE12})\frac{}{[\mathcal{M}]_C^c\langle\delta_p,\emptyset,\surd,q\rangle\xrightarrow{(\top,waiting,1)}\mathcal{M}_C^c\langle\delta_p,\emptyset,\surd,q\rangle}$$

Figure 4: Rules of governing policy enforcement

manager depends on the mode in which it is running. To distinguish managers in different modes, we use different notations. Let C denote the configuration set of manager $\mathcal{M}$, $c \stackrel{\text{def}}{=} \langle g, p\rangle \in C$ denote the current configuration, p_i denote the triggered adaptation policy, $g' \subseteq g$ is the set of triggered simple governing policies to be enforced, a is the action of a simple governing policy being executed by $\mathcal{M}$, and q is the input queue of $\mathcal{M}$. We denote manager $\mathcal{M}$ in the enforcement mode by $[\mathcal{M}]_C^c\langle\delta_p, g', a, q\rangle$. The notations $\mathcal{M}_C^c\langle p_i, \emptyset, \surd, q\rangle$, $|\mathcal{M}|_C^c\langle p_i, g', a, q\rangle$ and $\|\mathcal{M}\|_C^c\langle p_i, \emptyset, \surd, q\rangle$ indicate $\mathcal{M}$ in waiting, loose adaptation and strict adaptation modes, respectively. $\surd$ is a special constant to show termination of enforcing an action. The initial state of the manager is defined as $\mathcal{M}_C^{c_{init}}\langle\delta_p, \emptyset, \surd, \emptyset\rangle$ where c_{init} denotes the initial configuration of $\mathcal{M}$. The conditions of transition system are evaluated on the view layer.

Event Propagation. An event is a predicate evaluated on the transitions of the actor layer. Let $observe(e) \in \mathcal{A}$ be the primitive action for observing event e. Rule EPR asserts that when event e is satisfied by a transition at the actor layer, the message $observe(e)$ is appended to the input queue of all the managers. This rule gives the semantics of event propagation for the manager in the governing policy enforcement mode. Similar rules are defined for the waiting, loose adaptation and strict adaptation modes.

$$\text{EPR}\frac{s_a \xrightarrow{l} s'_a \quad (s_a, l, s'_a) \vDash e \quad \mathcal{M} \in M}{[\mathcal{M}]^c_C \langle \delta_p, g', a, q\rangle \xrightarrow{l} [\mathcal{M}]^c_C \langle \delta_p, g', a, q : \mathit{observe}(e)\rangle}$$

Governing policy enforcement semantics. Whenever an event is received by a manager, it identifies all the triggered simple governing policies whose policy conditions evaluate to true and highest priority. Once a manager enforces all the triggered policies, it evolves to the waiting mode. Figure 4 gives the rules of governing policy enforcement. Due to the fact that in PobSAM adaptation has a higher priority than enforcing policies, we consider the priority of enforcing policies as "1".

Using NPE1, the manager switches to the enforcement mode by identifying the triggered policies ($g_i = \langle o_i, e_i, \psi_i\rangle \bullet a_i$) to be enforced in which $\text{Trig}(e, v)$ denotes the set of triggered policies due to the occurrence of event e in state v of the view layer. $\mathcal{T}_{g_i}(e)$ denotes the triggering condition of g_i when event e occurs. We will elaborate this function in Section 6.2. NPE2 places the action of a policy ($g_i.a$) in the action part of the manager to be run and removes g_i from the list of activated policies. NPE3 represents the execution of a primitive action α. NPE4 and NPE5 define the semantics of non-deterministic choice and NPE6 and NPE7 define the semantics of conditional actions, respectively. NPE8-11 apply sequential and parallel compositions of actions. When there is no policy to be enforced, the manager will switch to the waiting mode using NPE12 where $\mathit{waiting} \in \mathcal{A}$.

As mentioned above, managers in loose adaptation mode are able to enforce governing policies. Therefore, all the rules introduced for the enforcement mode, except for NPE12, are applicable in loose adaptation mode too. In loose adaptation mode, NPE1 is rewritten as rule LAE1:

$$(\text{LAE1})\frac{\text{Trig}(e,v)=\{g_i \in g \mid v \vDash \psi_i, e_i = e, \nexists g_j \in g. v \vDash \psi_j \wedge e_j = e \wedge o_j > o_i\} \qquad \text{Trig}(e,v) \neq \emptyset}{|\mathcal{M}|^c_C \langle p_i, \emptyset, \surd, q\rangle \xrightarrow{(\mathcal{T}_{g_i}(e), \mathit{observe}(e), 1)} |\mathcal{M}|^c_C \langle p_i, \text{Trig}(e,v), \surd, q\rangle}$$

Adaptation policy enforcement semantics. Figure 5 shows the rules for adaptation in strict mode. SAR1 states that the adaptation policy $\langle o, e, \psi, \lambda, \phi\rangle \bullet c$ is triggered with the priority $o + 1$, when event e occurs and the condition ψ holds. If the adaptation type of that policy is strict adaptation type, the manager $\mathcal{M}$ switches to the strict adaptation mode by performing primitive action $\mathit{tostrict}(e) \in \mathcal{A}$. SAR2 asserts that when the condition for applying the adaptation holds, $\mathcal{M}$ will evolve to the waiting mode by performing primitive action $\mathit{switch}(c) \in \mathcal{A}$, and run configuration c'. Rules of loose adaptation are similar to the strict adaptations rules defined as follows where $\mathit{toloose}(e) \in \mathcal{A}$:

$$(\text{LAR1})\frac{c = \langle g, p\rangle \quad p = p_1 \oplus p' \quad p_1 = \langle o, e, \psi, \lambda, \phi\rangle \bullet c' \quad v \vDash \psi \quad \lambda = \bot}{\mathcal{M}^c_C \langle \delta_p, \emptyset, \surd, \mathit{observe}(e) : q\rangle \xrightarrow{(\psi, \mathit{toloose}(e), o+1)} |\mathcal{M}|^c_C \langle p_1, \emptyset, \surd, q\rangle}$$

$$(\text{LAR2})\frac{p_1 = \langle o, e, \psi, \lambda, \phi\rangle \bullet c' \quad v \vDash \phi}{|\mathcal{M}|^c_C \langle p_1, \emptyset, \surd, q\rangle \xrightarrow{(\phi, \mathit{switch}(c'), o+1)} \mathcal{M}^{c'}_C \langle \delta_p, \emptyset, \surd, q\rangle}$$

$$(\text{SAR1})\frac{c = \langle g, p\rangle \quad p = p_1 \oplus p' \quad p_1 = \langle o, e, \psi, \lambda, \phi\rangle \bullet c' \quad v \vDash \psi \quad \lambda = \top \quad c \neq c'}{\mathcal{M}^c_C\langle \delta_p, \emptyset, \surd, observe(e) : q\rangle \xrightarrow{(\psi, tostrict(e), o+1)} \|\mathcal{M}\|^c_C\langle p_1, \emptyset, \surd, q\rangle}$$

$$(\text{SAR2})\frac{p_1 = \langle o, e, \psi, \lambda, \phi\rangle \bullet c' \quad v \vDash \phi}{\|\mathcal{M}\|^c_C\langle p_1, \emptyset, \surd, q\rangle \xrightarrow{(\phi, switch(c'), o+1)} \mathcal{M}^{c'}_C\langle \delta_p, \emptyset, \surd, q\rangle}$$

Figure 5: Rules of strict adaptation

$$(\text{DRR1})\frac{c' \in C \qquad p_1 = \langle 2, e, \top, \lambda, \phi\rangle \bullet c' \quad \lambda = \bot \quad c \neq c'}{\mathcal{M}^c_C\langle \delta_p, \emptyset, \surd, load(c', \lambda, \phi) : q\rangle \xrightarrow{(\top, load(c', \lambda, \phi), 2)} |\mathcal{M}|^c_C\langle p_1, \emptyset, \surd, q\rangle}$$

$$(\text{DRR2})\frac{c' \in C \qquad p_1 = \langle 1, e, \top, \lambda, \phi\rangle \bullet c' \quad \lambda = \top \quad c \neq c'}{\mathcal{M}^c_C\langle \delta_p, \emptyset, \surd, load(c', \lambda, \phi) : q\rangle \xrightarrow{(\top, load(c', \lambda, \phi), 1)} \|\mathcal{M}\|^c_C\langle p_1, \emptyset, \surd, q\rangle}$$

$$(\text{DRR3})\frac{c \neq c' \qquad c \in C}{\mathcal{M}^c_C\langle \delta_p, \emptyset, \surd, remove(c') : q\rangle \xrightarrow{(\top, remove(c'), 1)} \mathcal{M}^c_{C\backslash c'}\langle \delta_p, \emptyset, \surd, q\rangle}$$

$$(\text{DRR4})\frac{c' \notin C}{\mathcal{M}^c_C\langle \delta_p, \emptyset, \surd, add(c') : q\rangle \xrightarrow{(\top, add(c'), 1)} \mathcal{M}^c_{C\cup c'}\langle \delta_p, \emptyset, \surd, q\rangle}$$

Figure 6: Rules of dynamic adaptation of configurations

The semantics of dynamic configurations. Figure 6 shows the semantics of dynamic adaptation of configurations. When the manager dequeues a message $load(c', \lambda, \phi)$ to load configuration c', it switches to either strict or loose adaptation modes according to the value of λ (DRR1, DRR2). When condition ϕ becomes true, it switches to configuration c' (LAR2 or SAR2). Rule DRR3 removes configuration $c' \in C$ from the configuration set, provided that c' is not the current configuration of manager. Rule DRR4 adds a new configuration c' to the configuration set of manager where $c' \notin C$.

Table 1 shows the rules applied in each mode.

6. Behavioral Equivalence

In this section, we provide an equational theory to reason about behavioral equivalence of managers, configurations and policies. To this end, we present an axiom system for CA^a (the algebra for actions of simple governing policies), modulo splitting bisimilarity. We introduce an operator which formulates the behavior of a governing policy set in terms of CA^a terms. Then we use the axiom system of CA^a, to reason about behavioral equivalence of governing policy sets. We extend splitting bisimulation with priorities called prioritized splitting

mode	rules
waiting	EPR, NPE1, SAR1, LAR1, DRR1-4
strict adaptation	EPR, SAR2
loose adaptation	EPR, LAR2, LAE1-11
enforcement	EPR, NPE2-12

Table 1: Rules applied in different modes

bisimulation, and introduce an axiom system for CA^p (the algebra for expressing adaptation policies), modulo prioritized splitting bisimilarity. The behavioral equivalence of configurations is defined based on the behavioral equivalence of their governing policy set and adaptation policies. Furthermore, we present the behavioral equivalence of managers based on the behavioral equivalence of their configurations.

This behavioral equivalence theory is used to reason about PobSAM models. One of the main results of our equational theory is compositionality, i.e. preserving the semantics of the managed actor system by substitution of a policy, configuration or manager by an equivalent one. The messages sent to the actors by the managers of each state are consequences of enforcing governing policies. We will show that two behavioral equivalent managers enforce equivalent governing policy sets, and subsequently, they send the same sequences of messages to the actors. Therefore, when a manager such as m_1 is replaced by an equivalent manager like m_2, the managed actors controlled by m_1 behave identically to the case that they are governed by m_2.

6.1. Prioritized Splitting Bisimulation

In this section, we introduce the notion of *prioritized splitting bisimulation* to define the equivalence of two prioritized conditional state transition systems. In a prioritized conditional state transition system, when the condition of a transition t_1 from state s implies disjunction the conditions of transitions with higher priorities from s, then transition t_1 is never executed:

Definition 2. Let $\mathrm{T} = \langle S, \rightarrow, \rightarrow_{\surd}, s^0 \rangle$ denote a prioritized conditional state transition system. We call a transition $s \xrightarrow{(\phi,\alpha,n)} s' \in \rightarrow$ (or $s \xrightarrow{(\phi,\alpha,n)} \surd \in \rightarrow_{\surd}$) non-triggerable iff there are transitions t_i from s, $1 \leq i \leq m$, where $t_i = s \xrightarrow{(\phi_i,\alpha_i,n_i)} s_i$ or $t_i = s \xrightarrow{(\phi_i,\alpha_i,n_i)} \surd$, $n_i > n$ and $\phi \rightarrow \bigvee \phi_i$. A triggerable transition is a transition which is not non-triggerable.

Furthermore, the function $O(s, n)$ gives the relative priority of triggerable transitions from $s \in S$ with priority $n \in \mathbb{N}$, with respect to all triggerable transitions from s. If there is no triggerable transition with priority n from state s, then $O(s, n) = 0$.

Example 7. The transition $s_0 \xrightarrow{(\psi_1,\alpha,2)} s_2$ of prioritized conditional state transitions shown in Figure 7 is non-triggerable. It is covered by the transition

$s_0 \xrightarrow{(\phi_1,\alpha,4)} s_1$ where $\psi_1 \implies \phi_1$. Furthermore, $O(s_0,4) = 1$ and $O(s_0,n) = 0$ where $n \neq 4$.

The prioritized conditional transitions represent the execution of the transition at any state satisfying the condition with the highest priority among enabled transitions. We can use several transitions from that state to cover all the cases that satisfy the corresponding condition. In prioritized splitting bisimulation, the conditions of a transition with relative priority n from state s of one of the related transition systems may be simulated by several transitions with identical relative priorities from the corresponding state in the other transition system.

Definition 3. Prioritized Splitting Bisimulation Let $\mathrm{T}_1 = \langle S_1, \to_1, \to_{\surd_1}, s_1^0 \rangle$ and $\mathrm{T}_2 = \langle S_2, \to_2, \to_{\surd_2}, s_2^0 \rangle$ denote two prioritized conditional state transition systems. A binary relation $\mathcal{R} \subseteq S_1 \times S_2$ is a prioritized splitting bisimulation iff $(s_1^0, s_2^0) \in \mathcal{R}$ and $\forall_{s_1,s_2} (s_1, s_2) \in \mathcal{R} \Rightarrow$:

- for each triggerable transition $s_1 \xrightarrow{(\phi,\alpha,n)}_1 s_1^{'}$, there exists a finite set $CS' \subseteq \mathcal{B}^- \times S_2 \times \mathbb{N}$ such that $\bigvee_{(\phi',s_2',n') \in CS', O(s_1,n)=O(s_2,n')} \phi' \to \phi$, $s_2 \xrightarrow{(\phi',\alpha,n')}_2 s_2^{'}$ and $(s_1^{'}, s_2^{'}) \in \mathcal{R}$ for all $(\phi', s_2', n') \in CS'$;
- for each triggerable transition $s_2 \xrightarrow{(\phi,\alpha,n)}_2 s_2^{'}$, there exists a finite set $CS' \subseteq \mathcal{B}^- \times S_1 \times \mathbb{N}$ such that $\bigvee_{(\phi',s_1',n') \in CS', O(s_2,n)=O(s_1,n')} \phi' \to \phi$, $s_1 \xrightarrow{(\phi',\alpha,n')}_1 s_1^{'}$ and $(s_1^{'}, s_2^{'}) \in \mathcal{R}$ for all $(\phi', s_1', n') \in CS'$;
- if $s_1 \xrightarrow{(\phi,\alpha,n)} \surd_1$ is triggerable, then there is a set $C' \subseteq \mathcal{B}^- \times \mathbb{N}$ such that $\bigvee_{(\phi',n') \in C', O(s_2,n)=O(s_1,n')} \phi' \to \phi$ and for all $(\phi', n') \in C'$, $s_2 \xrightarrow{(\phi',\alpha,n')} \surd_2$;
- if $s_2 \xrightarrow{(\phi,\alpha,n)} \surd_2$ is triggerable, then there is a set $C' \subseteq \mathcal{B}^- \times \mathbb{N}$ such that $\bigvee_{(\phi',n') \in C', O(s_1,n)=O(s_2,n')} \phi' \to \phi$ and for all $(\phi', n') \in C'$, $s_1 \xrightarrow{(\phi',\alpha,n')} \surd_1$;

Two prioritized conditional state transition systems T_1 and T_2 are prioritized splitting bisimilar, denoted by $T_1 \underline{\Leftrightarrow}_p T_2$ if and only if there is a prioritized splitting bisimulation relation such as $\mathcal{R}$ between T_1 and T_2. Let $\mathcal{R}$ be a prioritized splitting bisimulation between T_1 and T_2, then we say $\mathcal{R}$ is a prioritized splitting bisimulation witnessing $T_1 \underline{\Leftrightarrow}_p T_2$.

Example 8. The prioritized conditional state transition systems given in figure 7 are prioritized splitting bisimilar. In this figure, the dashed lines indicate a prioritized splitting bisimulation relation between two transition system.

A conditional state transition system is a prioritized state transition system in which all transitions have the same priority. Splitting bisimulation [10] is used to define behavioral equivalence of conditional state transition systems. In splitting bisimulation, a transition of a transition system is simulated by several transition of another transition system. It is too trivial to prove that two conditional state transition systems are splitting bisimilar, if and only if their corresponding prioritized state transition systems are prioritized splitting bisimilar.

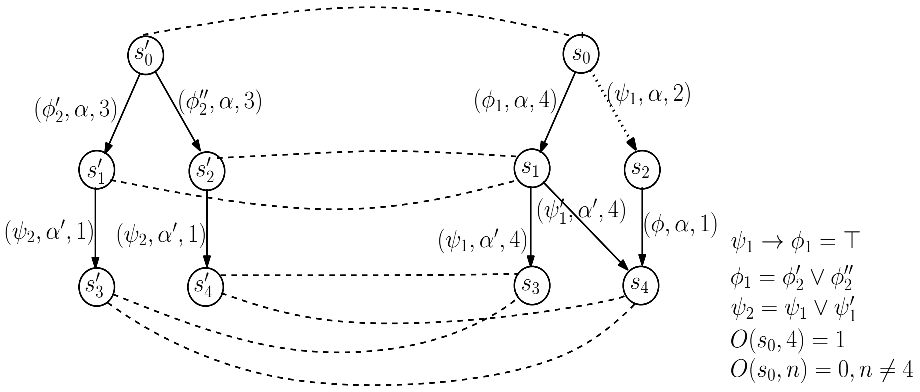


Figure 7: Prioritized Splitting Bisimulation

6.2. Behavioral Equivalence of Governing Policies

We introduce an axiomatization for CA^a, modulo splitting bisimulation. The axioms presented in Table 2 constitute the axiom system of CA^a taken from [10]. These axioms describe the basic identities between terms in A. The operator $+$ is commutative, associative and idempotent (A1-A3). δ_a behaves as the neutral element for $+$(A4). Furthermore, the operator ; right-distributes over $+$ and is associative (A6,A7). C1-C8 are axioms defined for the conditional-choice operator. AP1-AP6 are standard axioms of the operators $\|$ and $\lfloor\!\lfloor$ for PA$_\delta^{cc}$. Congruence of splitting bisimulation, and soundness and ground-completeness of our axiom system follow from the corresponding theorems of PA$_\delta^{cc}$ [10]. Moreover, the models of CA^a terms are defined in terms of conditional state transition systems in [10].

Now, we proceed to present the behavioral equivalence on the governing policy sets. A simple governing policy is a set of actions which must be enforced in the system under specific circumstances. In order to reason about the behavioral equivalence of governing policy sets, first we define an operator Ψ which describes the behavior of a governing policy set as an action term. Given $\Psi(g)$ and $\Psi(g')$ of two arbitrary governing policy sets g and g', we use the axiom system of CA^a to check the equivalence of $\Psi(g)$ and $\Psi(g')$.

When an event occurs, the triggered policies with the highest priorities are chosen to be enforced in the system. The actions of the triggered policies will be sequentially executed in an arbitrary order. As an example consider the policy set $g = \{g_1, g_2, g_3\}$ where $g_1 = \langle 1, e_1, \phi_1 \rangle \bullet a_1$, $g_2 = \langle 1, e_2, \phi_2 \rangle \bullet a_2$, $g_3 = \langle 1, e_1, \phi_3 \rangle \bullet a_3$ and $e_1 \neq e_2$. When event e_1 occurs, four cases can occur:

(i) if only the conditions of g_1 become true (i.e., $\phi_1 \wedge \neg\phi_3 = \top$), then a_1 is executed,

(ii) if only the conditions of g_3 become true (i.e., $\phi_3 \wedge \neg\phi_1 = \top$), then a_3 is executed,

(iii) if the conditions of both policies becomes true (i.e., $\phi_1 \wedge \phi_3 = \top$), then

$$\begin{array}{ll}
a + a' = a' + a & \text{A1} \\
(a + a') + a'' = a + (a' + a'') & \text{A2} \\
a + a = a & \text{A3} \\
a + \delta_a = a & \text{A4} \\
\delta_a ; a = \delta_a & \text{A5} \\
(a + a'); a'' = a; a'' + a'; a'' & \text{A6} \\
(a; a'); a'' = a; (a'; a'') & \text{A7}
\end{array}
\qquad
\begin{array}{ll}
\top :\to a = a & \text{C1} \\
\bot :\to a = \delta_a & \text{C2} \\
\phi :\to (a + a') = \phi :\to a + \phi :\to a' & \text{C3} \\
\phi :\to (a; a') = \phi :\to a; a' & \text{C4} \\
\phi :\to (\psi :\to a) = (\phi \wedge \psi) :\to a & \text{C5} \\
(\phi \vee \psi) :\to a = \phi :\to a + \psi :\to a & \text{C6} \\
\phi :\to \delta_a = \delta_a & \text{C7} \\
\phi :\to a \lfloor\!\lfloor a' = \phi :\to (a \lfloor\!\lfloor a') & \text{C8}
\end{array}$$

$$\begin{array}{ll}
a \parallel a' = a' \parallel a & \text{AP1} \\
(a \parallel a') \parallel a'' = a \parallel (a' \parallel a'') & \text{AP2} \\
(a + a') \lfloor\!\lfloor a'' = (a \lfloor\!\lfloor a'') + (a' \lfloor\!\lfloor a'') & \text{AP3} \\
a \parallel a' = a \lfloor\!\lfloor a' + a' \lfloor\!\lfloor a & \text{AP4} \\
\alpha \lfloor\!\lfloor a = \alpha; a & \text{AP5} \\
(\alpha; a) \lfloor\!\lfloor a' = \alpha; (a \parallel a') & \text{AP6}
\end{array}$$

Table 2: Action Algebra CA^a

either the action $a_1; a_3$ or the action $a_3; a_1$ is performed,

(iv) if neither the conditions of g_1 nor the conditions of g_3 becomes true, then action δ_a is performed. Therefore, the behavior of this governing policy set is formalized as the following action, when event e_1 occurs:

$$\begin{aligned}
a \quad = \quad & (\phi_1 \wedge \neg\phi_3 :\to a_1) \ + \ (\neg\phi_1 \wedge \phi_3 :\to a_3) \ + \\
& (\phi_1 \wedge \phi_3 :\to (a_1; a_3 + a_3; a_1)) + (\neg\phi_1 \wedge \neg\phi_3 :\to \delta_a)
\end{aligned}$$

Now, we proceed by formulating the behavior of a governing policy set, when an event such as e occurs. Given the governing policy set $g = \{g_1, ..., g_n\}$, the operator $\Psi(g, e)$ returns the action terms (in A) that are enforced by a manager due to a triggered set of governing policies ($g' \subseteq g$), when an event ($e \in E$) occurs. Formula $\mathcal{T}_{g_i}$ defines the activation condition of $g_i = \langle o_i, e_i, \phi_i \rangle \bullet a_i$ when event e occurs as follows:

$$\mathcal{T}_{g_i}(e) \equiv \begin{cases} \phi_i \wedge \neg \bigvee_{o_i < o_k, e_k = e} \phi_k & e_i = e \\ \bot & e_i \neq e \end{cases} \tag{2}$$

which informally asserts that g_i is triggered, if g_i's event is e, its condition is true and no other policy with a higher priority is triggered. Assume g' indicates the set of triggered simple governing policies. $Z(g')$ is a function which gives the choice between all the permutations of the actions of simple policies in g', i.e., this function gives different strategies to enforce the triggered policies. For example, in the third case of above example (iii), $g' = \{g_1, g_3\}$ and $Z(g') = \{a_1; a_3\ ,\ a_3; a_1\}$. The operator $\Psi : g \times E \to A$ is defined as follows:

$$\Psi(g, e) = \mathit{observe}(e); \sum_{g' \subseteq g} \phi(g', e) :\to Z(g') \qquad \text{GA1}$$

in which $\phi(g', e) = \bigwedge_{g_i \in g'} \mathcal{T}_{g_i}(e) \ \wedge \ \bigwedge_{g_i \in (g-g')} \neg \mathcal{T}_{g_i}(e),\ g_i \in g$. The function $\phi(g', e)$ gives the conditions of triggering policy set g' when event e occurs, e.g., in the above example $\phi(\{g_1, g_3\}, e_1) = \phi_1 \wedge \phi_3$ and $\phi(\{g_1\}, e_1) = \phi_1 \wedge \neg\phi_3$. It is clear that the behavior of a governing policy set g when event e occurs, is equal to the behavior of a governing policy set g_e where $g^e = \{g_i \in g | e_i = e\}$, i.e.,

$$\Psi(g, e) = \Psi(g^e, e) \qquad\qquad \text{GA2}$$

The behavior of a governing policy set is the choice of its behavior for each event, i.e., the behavior of a governing policy set such as g in terms of CA^a terms is formulated as follows:

$$\Psi(g) = \sum_{e \in E} \Psi(g, e) \qquad\qquad \text{GA3}$$

Example 9. When the wounded person is found by a surveyor, a "success" message is sent to the commander using the following policies. When the surveyor has enough energy, it will send the message to $relay_2$, otherwise the message is sent to the commander through $relay_1$. The following simple governing policies are used to specify this situation:

$$\begin{aligned} g_2 &\overset{\text{def}}{=} \langle 2,\ found(x, y),\ enoughEnergy\rangle \bullet \mathit{relay2.send}(msg, commander) \\ g_3 &\overset{\text{def}}{=} \langle 1,\ found(x, y),\ \top\rangle \bullet \mathit{relay1.send}(msg, commander) \end{aligned}$$

Let g denote the governing policy set of surveyor where only the policy set $\{g_2, g_3\}$ can be triggered when event *found* occurs. The conditions of triggering g_2 and g_3, when event $found$ occurs, are $\mathcal{T}_{g_2}(found(x, y)) = enoughEnergy$ and $\mathcal{T}_{g_3}(found(x, y)) = \neg enoughEnergy$, respectively. The conditions of triggering different subsets of g (i.e., g') are described as follows:

$$\phi(g', found(x, y)) = \begin{cases} enoughEnergy & g' = \{g_2\} \\ \neg enoughEnergy & g' = \{g_3\} \\ \bot & \text{otherwise} \end{cases}$$

Thus, the behavior of g when event $found(x, y)$ occurs is expressed using GA2 as follows:

$$\begin{aligned} \Psi(g, found(x, y)) \quad = \quad & observe(found(x, y)); \\ & (\neg enoughEnergy :\rightarrow \mathit{relay1.send}(msg, commander)) + \\ & (enoughEnergy :\rightarrow \mathit{relay2.send}(msg, commander)) + \\ & \sum_{g' \subseteq g, g' \neq \{g_2\}, g' \neq \{g_3\}} \bot :\rightarrow Z(g') \end{aligned}$$

Definition 4. We say two actions a and a' are splitting bisimilar, denoted by $a \underline{\Leftrightarrow} a'$, iff the models of a and a' in terms of conditional state transitions systems, defined in [10], are splitting bisimilar.

Corollary 1. ***(Soundness)*** *The axioms GA1-3 are sound for splitting bisimulation equivalence, i.e., for all policy sets g, g' of CA^g and $e \in E$, (i) $\Psi(g) = \Psi(g')$ implies $\Psi(g) \underline{\Leftrightarrow} \Psi(g')$, and (ii) $\Psi(g,e) = \Psi(g',e)$ implies $\Psi(g,e) \underline{\Leftrightarrow} \Psi(g',e)$.*

Proof. Since operator Ψ is a closed term from CA^a, soundness of GA1-3 follows from the soundness theorem of CA^a following [10].

Corollary 2. ***(Ground Completeness)*** *For all closed terms g, g' of CA^g and $e \in E$ (i) $\Psi(g) \underline{\Leftrightarrow} \Psi(g')$ implies $\Psi(g) = \Psi(g')$, and (ii) $\Psi(g,e) \underline{\Leftrightarrow} \Psi(g',e)$ implies $\Psi(g,e) = \Psi(g',e)$.*

Proof. Since operator Ψ is a closed term from CA^a, ground completeness of GA1-3 follows from the ground-completeness theorem of CA^a following [10].

Example 10. Consider a situation where $surveyor_1$ with the governing policy set $g = g'' \cup \{g_1\}$, with the g_1 defined in Example 4, is incapable to act as a surveyor. This UAV must be replaced by another UAV with the same behavior. To this end, the UAV $surveyor_2$ with the governing policy set $g' = g'' \cup \{g_3, g_4\}$ could be a candidate to replace $surveyor_1$, where g_3 is defined in examples 9 and g_4 is defined as follows:

$$g_4 \overset{\text{def}}{=} \langle 1,\ found(x,y),\ \top\rangle \bullet \mathit{BSN.getHealthInfo}()$$

Suppose when event $found$ occurs, none of the policies in g'' is triggered. The action terms due to enforcing g and g' are as follows:

$$\begin{aligned}
\Psi(g) &= \sum_{e} \Psi(g,e) = \Psi(g,\mathit{found}) + \sum_{e \neq found} \Psi(g,e) \overset{\text{GA2}}{=} \\
&\quad \Psi(g_1,\mathit{found}) + \sum_{e \neq found} \Psi(g'',e) \\
\Psi(g') &= \Psi(\{g_3,g_4\},\mathit{found}) + \sum_{e \neq found} \Psi(g'',e)
\end{aligned}$$

where

$$\begin{aligned}
\Psi(g_1,\mathit{found}) &= \mathit{observe}(found); \\
&\quad (\mathit{BSN.getHealthInfo}() \parallel \mathit{relay1.send}(\mathit{msg}, \mathit{commander})) \\
\Psi(\{g_3,g_4\},\mathit{found}) &= \mathit{observe}(found); \\
&\quad (\mathit{BSN.getHealthInfo}(); \mathit{relay1.send}(\mathit{msg}, \mathit{commander}) + \\
&\quad \mathit{relay1.send}(\mathit{msg}, \mathit{commander}); \mathit{BSN.getHealthInfo}())
\end{aligned}$$

It is straightforward to prove that $\Psi(g_1, \mathit{found})$ and $\Psi(\{g_3,g_4\}, \mathit{found})$ are equivalent according to CA^a axioms (AP4 and AP5). Consequently, $surveyor_2$ has the same behavior as $surveyor_1$ to act as a surveyor.

$$
\begin{array}{lll}
p \oplus p & = p & \text{PA1} \\
p \oplus p' & = p' \oplus p & \text{PA2} \\
p \oplus (p' \oplus p'') & = (p \oplus p') \oplus p'' & \text{PA3} \\
p \oplus \delta_p & = p & \text{PA4} \\
\langle o, e, \bot, \lambda, \phi \rangle \bullet c & = \delta_p & \text{PA5}
\end{array}
$$

$$\langle o, e, \psi, \lambda, \phi \rangle \bullet c \oplus \langle o, e, \psi', \lambda, \phi \rangle \bullet c = \langle o, e, \psi \vee \psi', \lambda, \phi \rangle \bullet c \qquad \text{PA6}$$

$$\langle o, e, \psi, \lambda, \phi \rangle \bullet c \oplus \langle o', e, \psi', \lambda', \phi' \rangle \bullet c' = \langle o, e, \psi, \lambda, \phi \rangle \bullet c$$
$$\text{if } o > o' \wedge \psi \rightarrow \psi' \qquad \text{PA7}$$

Table 3: Adaptation Policy Algebra CA^p

6.3. Behavioral Equivalence of Adaptation Policies

In this section, we introduce an axiom system, modulo prioritized splitting bisimulation to reason about the behavioral equivalence of adaptation policies. Table 3 shows the axioms of algebra CA^p in which p, p' and p'' are variables of sort **P**. The operator $\oplus$ is idempotent, commutative and associative (PA1-PA3). δ_p behaves as neutral element for $\oplus$ (PA4). PA5 describes when the condition of triggering an adaptation policy never holds, it acts as a null adaptation policy. An adaptation policy never becomes activated, provided that its triggering conditions imply the triggering conditions of another adaptation policy with a higher priority and identical event (PA7).

We presented the structural operational semantics of a manager in Section 5 which expresses the whole behavior of a manager. To reason about the behavioral equivalence of two adaptation policies, first we define the semantics of adaptation polices solely. We present the models of adaptation policies (CA^p) in terms of prioritized conditional state transition systems. A model of an algebra such as CA^p indicates its semantics which is a structure that consists of (1) a non-empty set D, called the domain of the model, (2) for each constant of CA^p an element of D and (3) for each n-ary operator of CA^p, an n-ary operation on D.

CA^p's models are obtained by associating an element of $\mathbb{PCTS}$ for the constant δ_p and each simple adaptation policy p_i, and an operation on $\mathbb{PCTS}$ corresponding to the operator $\oplus$. The constant δ_p is associated to constant $\hat{\delta_p}$, the simple adaptation policy p_i is associated to $\hat{p_i}$, and operator $\oplus$ is associated to operator $\hat{\oplus}$ of $\mathbb{PCTS}$. An arbitrary set $\mathcal{S}$ closed under $\uplus$ is assumed where $S \uplus S'$ is the disjoint union of the sets S and S' defined as $S \uplus S' = (S \times \{\emptyset\}) \cup (S' \times \{\{\emptyset\}\})$. It is worth mentioning that the identity of the states of a prioritized conditional transition system is not relevant to the behavior represented by it. The models of CA^p are obtained as follows:

- $\hat{\delta_p} = \langle \{s_0\}, \emptyset, \emptyset, s_0 \rangle$ where $s_0 \in \mathcal{S}$.
- Let s_0 and s_1 be distinct members of $\mathcal{S}$. The model of a simple adaptation policy $p_i = \langle o, e, \psi, \lambda, \phi \rangle \bullet c$ is defined as $\hat{p_i} = \langle S, \rightarrow, \rightarrow_{\surd}, s_0 \rangle \in \mathbb{PCTS}$ where

$S = \{s_0, s_1\}$:

$$\begin{array}{rcl} \xrightarrow{(\psi,tostrict(e),o+1)} & = & \{(s_0, s_1) | \lambda = \top\}, \\ \xrightarrow{(\psi,toloose(e),o+1)} & = & \{(s_0, s_1) | \lambda = \bot\}, \\ \xrightarrow{(\phi,switch(c),o+1)}\surd & = & \{s_1\}, \end{array}$$

- Let $\mathrm{T}_p = \langle S, \rightarrow, \rightarrow_{\surd}, s_0\rangle \in \mathbb{PCTS}$ and $\mathrm{T}_{p'} = \langle S', \rightarrow', \rightarrow'_{\surd}, s'_0\rangle \in \mathbb{PCTS}$ indicate the models of two adaptation policies p and p', respectively. Then, $T_p \hat{\oplus} T_{p'} = \langle S'', \rightarrow'', \rightarrow''_{\surd}, s''_0\rangle$ where $s''_0 \in \mathcal{S} \backslash (S \uplus S')$, $S'' = \{s''_0\} \cup (S \uplus S')$, $\mu(s) = (s, \emptyset)$, $\mu'(s) = (s, \{\emptyset\})$, and $\forall (\phi, \alpha, n) \in \mathcal{B}^- \times \mathcal{A} \times \mathbb{N}$:

$$\begin{array}{rcl} \xrightarrow{(\phi,\alpha,n)}'' & = & \{(s''_0, \mu(s_1)) | s_0 \xrightarrow{(\phi,\alpha,n)} s_1\} \cup \{(\mu(s_1), \mu(s_2)) | s_1 \xrightarrow{(\phi,\alpha,n)} s_2\} \\ & \cup & \{(s''_0, \mu'(s'_1)) | s'_0 \xrightarrow{(\phi,\alpha,n)}' s'_1\} \cup \{(\mu'(s'_1), \mu'(s'_2)) | s'_1 \xrightarrow{(\phi,\alpha,n)}' s'_2\} \\ \xrightarrow{(\phi,\alpha,n)}'' \surd & = & \{\mu(s) | s \xrightarrow{(\phi,\alpha,n)} \surd\} \\ & \cup & \{\mu'(s) | s \xrightarrow{(\phi,\alpha,n)}' \surd\}, \end{array}$$

The full prioritized splitting bisimulation models $\aleph$ of CA^p are the models whose domain is equivalence class of prioritized conditional state transition systems modulo prioritized splitting bisimulation. $\aleph$ is the expansion of $\mathcal{B}$ with (i) the non-empty set P for the sort $\mathbf{P}$, (ii) $\tilde{\delta_p} \in P$ for the constant δ_p, (iii) $\tilde{p} \in P$ for the simple adaptation policy p, and (iv) $\tilde{\oplus} : P \times P \rightarrow P$ for the operator $\oplus$ which are defined as follows:

$$\begin{array}{rcl} P & = & \mathbb{PCTS}/\Leftrightarrow_p, \\ \tilde{\delta_p} & = & [\hat{\delta_p}]_{\Leftrightarrow_p} \\ \tilde{p} & = & [\hat{p}]_{\Leftrightarrow_p} \\ [Z_1]_{\Leftrightarrow_p} & \tilde{\oplus} & [Z_2]_{\Leftrightarrow_p} = [Z_1 \hat{\oplus} Z_2]_{\Leftrightarrow_p} \end{array}$$

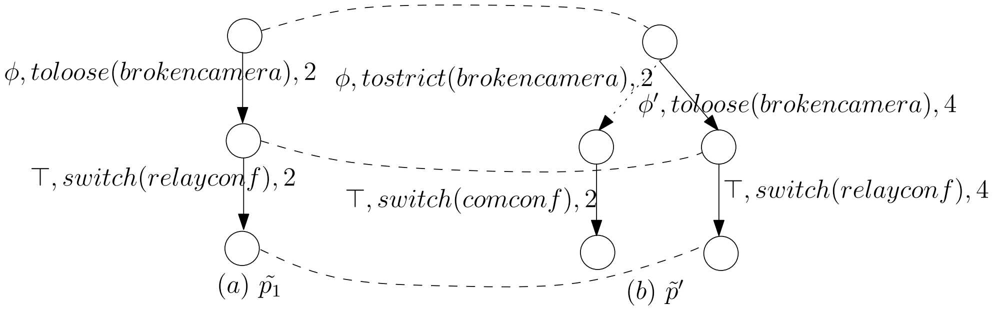
Definition 5. We say two adaptation policies p and p' are prioritized splitting bisimilar, denoted by $p \,\underline{\Leftrightarrow}_p\, p'$, iff their models in terms of prioritized conditional state transitions systems are prioritized splitting bisimilar.

Proposition 1. ***(Congruence)*** *Prioritized splitting bisimulation is a congruence with respect to $\oplus$, i.e., for all CA^p terms $p, p', q,$ and q', $p \,\underline{\Leftrightarrow}_p\, p'$ and $q \,\underline{\Leftrightarrow}_p\, q'$ imply $p \oplus q \,\underline{\Leftrightarrow}_p\, p' \oplus q'$.*

Proof. See Appendix A.

Theorem 1. ***(Soundness)*** *CA^p is sound for prioritized splitting bisimulation equivalence, i.e., for all CA^p terms p and p', $p = p'$ implies $p \,\underline{\Leftrightarrow}_p\, p'$.*

Proof. See Appendix A.


$\phi, toloose(brokencamera), 2$
$\phi, tostrict(brokencamera), 2$
$\phi', toloose(brokencamera), 4$
$\top, switch(relayconf), 2$
$\top, switch(comconf), 2$
$\top, switch(relayconf), 4$
$(a)\ \tilde{p_1}$
$(b)\ \tilde{p'}$

$$\phi = \neg success \wedge enoughEnergy$$
$$\phi' = \neg success$$

Figure 8: The model of the adaptation policy of example 5

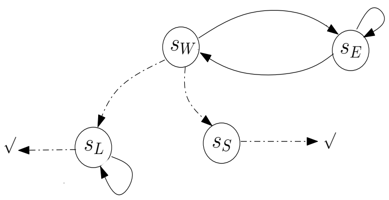


Figure 9: An Abstraction of the Behavior of a Configuration

Theorem 2. ***(Ground Completeness)*** *CA^p is ground complete for prioritized splitting bisimulation equivalence, i.e., for all CA^p terms p and p', $p \underline{\Leftrightarrow}_p p'$ implies $p = p'$.*

Proof. See Appendix A.

Example 11. Figures 8 (a) shows the model of adaptation policy p_1 in example 5, and Figure 8 (b) gives the model of adaptation policy p' defined as follows:

$$\begin{aligned} p' \quad = \quad & \langle 3, brokencamera, \neg success, \bot, \top\rangle \bullet relayconf \oplus \\ & \langle 1, brokencamera, \neg success \wedge enoughEnergy, \top, \top\rangle \bullet comconf \end{aligned}$$

The relation witnessing prioritized splitting bisimulation between p_1 and p' is shown in Figure 8 by dashed lines. Furthermore, we can prove $p_1 = p'$, according to axiom P7.

6.4. Behavioral Equivalence of Configurations

Let $c = \langle g, p\rangle$ denote a configuration with the adaptation policy p and the governing policy set g. Figure 9 gives an abstraction of the behavior of a configuration. In this figure, s_W, s_E, s_L and s_S indicate the abstract states of waiting mode, enforcement mode, loose adaptation mode and strict adaptation mode, respectively. The dashed dotted transitions model the transitions of adaptation policies while solid transitions model the transitions due to enforcing governing policies. The symbol $\surd$ denotes termination of executing configuration c. Note that s_L has a solid loop because of enforcing governing policies. Suppose $T_p = \langle S_p, \to_p, \to_{p\surd}, {s_p}^0\rangle \in \mathbb{PCTS}$ and $T_g = \langle S_g, \to_g, \to_{g\surd}, s_g^0\rangle$ indicate the models of p and g, respectively. Then, the model of c is $\Gamma(T_c = \langle S, \to, \to_{\surd}, s_0\rangle \in \mathbb{PCTS})$ where $S = S_m \times S_g \times S_p$, $S_m = \{s_S, s_W, s_E, s_L\}$, and the members of S_m are distinct members of $\mathcal{S}\backslash(S_p \cup S_g)$. Let $\mathcal{A}_g$ and $\mathcal{A}_p$ indicate the action sets of T_g and T_p, respectively. Furthermore, $\Gamma(T_c)$ denotes the connected part of transition system T_c whose states are reachable from the initial state of T_c. The transition relation of T_c is defined as follows:

- *Transitions due to enforcing governing policies*

  For all $(\phi, \alpha) \in \mathcal{B}^- \times A_g$, the following relations are defined:

$$\begin{aligned}
\xrightarrow{\langle\phi,\alpha,1\rangle} \quad = \quad & \{((s_W, s_g^0, s_p^0), (s_E, s, s_p^0)) | s_g^0 \xrightarrow{(\phi,\alpha)}_g s\} \\
\cup \quad & \{((s_E, s, s_p^0), (s_E, s', s_p^0)) | s \xrightarrow{(\phi,\alpha)}_g s'\} \\
\cup \quad & \{((s_E, s, s_p^0), (s_W, s_g^0, s_p^0)) | s \xrightarrow{(\phi,\alpha)}_{g\surd}\} \\
\cup \quad & \{((s_L, s, s_p), (s_L, s', s_p)) | s \xrightarrow{(\phi,\alpha)}_g s' \wedge \exists_{\phi',\alpha',n}\ s_p \xrightarrow{\langle\phi',\alpha',n\rangle} \surd\} \\
\cup \quad & \{((s_L, s, s_p), (s_L, s_g^0, s_p)) | s \xrightarrow{(\phi,\alpha)}_g \surd \wedge \exists_{\phi',\alpha',n}\ s_p \xrightarrow{\langle\phi',\alpha',n\rangle} \surd\}
\end{aligned}$$

- *Transitions due to enforcing adaptation policies*

  For all $(\phi, \alpha, n) \in \mathcal{B}^- \times \mathcal{A}_p \times \mathbb{N}$, the following relations are defined:

$$\begin{aligned}
\xrightarrow{\langle\phi,\alpha,n\rangle} \quad = \quad & \{((s_W, s_g^0, s_p^0), (s_S, s_g^0, s)) | s_p^0 \xrightarrow{\langle\phi,\alpha,n\rangle}_p s \wedge \alpha = \mathit{tostrict}(e)\} \\
\cup \quad & \{((s_W, s_g^0, s), (s_L, s_g^0, s')) | s \xrightarrow{\langle\phi,\alpha,n\rangle}_p s' \wedge \alpha = \mathit{toloose}(e)\} \\
\xrightarrow{\langle\phi,\alpha,n\rangle} \surd \quad = \quad & \{(s_S, s_g^0, s) | s \xrightarrow{\langle\phi,\alpha,n\rangle}_p \surd\} \cup \{(s_L, s_g^0, s) | s \xrightarrow{\langle\phi,\alpha,n\rangle}_p \surd\}
\end{aligned}$$

We describe the behavioral equivalence of two configurations based on the behavioral equivalence of their governing policies as well as the behavioral equivalence of their adaptation policies:

Definition 6. Let $c = \langle g, p\rangle$ and $c' = \langle g', p'\rangle$ be two arbitrary configurations. $c = c'$ iff $\Psi(g) = \Psi(g')$ and $p = p'$. Furthermore, we say c and c' are prioritized splitting bisimilar, written by $c \underline{\Leftrightarrow}_p c'$, if there is a prioritized splitting bisimulation relation between the models of c and c'.

Theorem 3. *Let $c = \langle g, p\rangle$ and $c' = \langle g', p'\rangle$ be two arbitrary configurations. $c = c'$ iff $c \underline{\Leftrightarrow}_p c'$.*

Proof. See Appendix B.

$$\langle c_i \mid C\rangle = \langle g_i,\ t_i(\langle c_1 \mid C\rangle, ..., \langle c_n \mid C\rangle)\rangle \quad i = \{1, ..., n\} \quad RDP$$

$$\text{if} \quad c_i' = \langle g_i,\ t_i(c_1', ..., c_n')\rangle \text{ for } i = \{1, ..., n\}, \text{then}$$

$$c_i' = \langle c_i \mid C\rangle \quad i = \{1, ..., n\} \quad RSP$$

Table 4: Recursion axioms for configurations

Since adaptation policies are specified in terms of configurations, the definition of configurations is recursive. Let $C = \{c_i = \langle g_i, p_i\rangle | 1 \leq i \leq n\}$ denote a set of configurations where $p_i = t_i(c_1, ..., c_n)$, t_i is a function and c_i is a variable. A solution of C is an interpretation of c_is such that the equations of C are satisfied. The construct $\langle c_i \mid C\rangle$ is a constant indicating component c_i of a solution of C. We use two common recursion axioms given in table 4 for guarded specifications [9, 8]. RDP(*Recursive Definition Principle*) states that the constant $\langle c_i \mid C\rangle$ is a solution of the ith component of C. RSP (*Recursive Specification Principle*) asserts that the recursive specification $C = \{c_i = \langle g_i, p_i\rangle | 1 \leq i \leq n\}$ has at most one solution per configuration.

Example 12. Consider a UAV controller with the configuration set $\{survconf, hazardconf, relayconf\}$ where $survconf = \langle gs, ps\rangle$, $hazardconf = \langle gh, ph\rangle$ and $relayconf = \langle gr, pr\rangle$. The configurations $survconf$ and $hazardconf$ are defined for surveying areas with safe and chemicals areas, respectively. Assume the governing policy sets of both surveying configurations are equivalent due to special conditions of the area, i.e., $\Psi(gh) = \Psi(gs)$, and

$$\begin{aligned} ps &= \langle 1, e, \phi, \bot, \psi\rangle \bullet relayconf \oplus \langle 2, e', \phi', \bot, \psi'\rangle \bullet hazardvonf \\ ph &= \langle 1, e, \phi, \bot, \psi\rangle \bullet relayconf \oplus \langle 2, e', \phi', \bot, \psi'\rangle \bullet survconf \\ pr &= \langle 1, e'', \phi_r, \top, \psi_r\rangle \bullet hazardvonf \end{aligned}$$

We cannot prove $ps = ph$ using CA^p axioms. However, the equation $survconf = hazardconf$ is proved using RDP and RSP as well as CA^p axioms.

6.5. Behavioral Equivalence of Managers

Checking behavioral equivalence of two managers is the most important part of the behavioral equivalence theory. As mentioned above, a manager runs one of its configurations at a time, and switches between the configurations to perform dynamic adaptation. Therefore, the model of a manager is simply generated by connecting the models of its configurations , i.e., a state s of configuration c_i is connected to the initial state of configuration c_j, $c_i \neq c_j$, if there is a transition from s with action $switch(c_j)$.

Let $m = \langle V_m, C, c_{init}\rangle$ indicate a manager where $C = \{c_1, ..., c_{k_1}\}$. Let $T_{c_i} = \langle S_i, \rightarrow_i, \rightarrow_{i\surd}, {s_i}^0\rangle \in \mathbb{PCTS}$ for $i = 1, ..., k_1$ denote the model of c_i. The model of m is defined as $T_m = \langle S, \rightarrow, \rightarrow_{\surd}, s^0\rangle \in \mathbb{PCTS}$ where $S = \bigcup_{1 \leq i \leq m} S_i$

and $\forall(\phi, \alpha, n) \in \mathcal{B}^{-} \times \mathcal{A} \times \mathbb{N}$:

$$\xrightarrow{(\phi,\alpha,n)} \quad = \quad \bigcup_{1\leq i\leq n} \xrightarrow{(\phi,\alpha,n)}_i \;\cup\; \{(s, s_j^0) | s \in \xrightarrow{(\phi,\alpha,n)}_i \surd \wedge \alpha = \mathit{switch}(c_j)\}$$

$$\xrightarrow{(\phi,\alpha,n)} \surd \quad = \emptyset$$

We proceed to reason about the behavioral equivalence of managers based on the behavioral equivalence of their configurations. The managers in equivalent initial configurations enforce equivalent governing policy sets. They should also switch to equivalent configurations, say c_i and c_i', using their equivalent adaptation policies. Similarly, the managers in equivalent configurations c_i and c_i' must enforce equivalent governing policy sets, and switch to equivalent configurations say c_j and c_j', and so on. Therefore, two managers are behavioral equivalent iff they have equivalent initial configurations. It is clear that configurations which are never active do not influence in the behavior of managers.

Definition 7. Let $m = \langle V_m, C, c_{init}\rangle$ and $m' = \langle V_{m'}, C', c'_{init}\rangle$ be two managers with configuration sets $C = \{c_1, ..., c_{k_1}\}$ and $C' = \{c'_1, ..., c'_{k_2}\}$, initial configurations $c_{init} \in C$ and $c'_{init} \in C'$, and views V_m and $V_{m'}$, respectively. We say $m = m'$ iff $c_{init} = c'_{init}$. Furthermore, we say m and m' are prioritized splitting bisimilar, written by $m \underline{\Leftrightarrow}_p m'$, if there is a prioritized splitting bisimulation relation between the models of m and m'.

Theorem 4. *Let $m = \langle V_m, C, c_{init}\rangle$ and $m' = \langle V_{m'}, C', c'_{init}\rangle$ be two arbitrary managers. Then, $m = m'$ iff $m \underline{\Leftrightarrow}_p m'$.*

Proof. See Appendix C.

Since, two behavioral equivalent managers have equivalent configurations (Theorem 4), they enforce equivalent governing policy sets (Theorem 3). The messages sent to the actors to control their behavior are consequences of enforcing governing policies. Hence, equivalent managers send the same sequences of messages to the actors. As a result, when a manager is replaced by an equivalent manager, the managed actors controlled by two equivalent managers behave identically. Therefore, the semantics of the managed actors is preserved by substitution of a manager (or a policy or a configuration) by an equivalent one.

Example 13. Consider the UAV of example 12 with the capability to search areas with chemical hazards. The manager of this UAV is defined as $\mathit{survCntrlr} = \langle V_s, \{\mathit{survconf}, \mathit{hazardconf}, \mathit{relayconf}\}, \mathit{survconf}\rangle$, where configuration $\mathit{survconf} = \langle gs, ps\rangle$ is used to search areas without hazardous chemicals, $\mathit{hazardconf} = \langle gh, ph\rangle$ is used to search areas with hazards, and $\mathit{relayconf} = \langle gr, pr\rangle$ is used for acting as a relay. Consider a situation that $\mathit{surveyor}_1$ has to be replaced by a UAV with the manager $\mathit{survCntrlr}' = \langle V_s, \{\mathit{survconf}', \mathit{relayconf}\}, \mathit{survconf}'\rangle$ where $\mathit{survconf}' =$

Dimension	**Degree**	**PobSAM Degree**	**Definition**
evolution	static to dynamic	dynamic	whether the goals can change within the lifetime of the system
binding time	static, semi-dynamic, dynamic	dynamic	the point in time when the adaptive behavior is composed with the business logic of an application
adaptation source	external ,internal	external or internal	where is the source of adaptation?
type	parameter, functional, and structural	functional and structural	whether adaptation is related to the parameters, the behavior or the structure of the system
autonomy	autonomous to assisted (system or human)	autonomous	what is the degree of outside intervention during adaptation
organization	centralized to decentralized	decentralized (by different managers)	whether the adaptation is done by a single component or distributed amongst several components
scope	local to global	local	whether adaptation is localized or involves the entire system
triggering	event-trigger to time-trigger	event-trigger	whether the change that triggers adaptation is associated with an event or a time slot

Table 5: Modeling dimensions of PobSAM according to [3, 19]

$\langle gs', ps'\rangle$. Let configurations $survconf$ and $survconf'$ have equivalent governing policy sets, i.e., $\Psi(gs) = \Psi(gs')$. The adaptation policies of $survconf$ and $survconf'$ are defined as follows where $\phi' \Rightarrow \phi$

$$\begin{aligned} ps &= \langle 1, e, \phi, \perp, \psi\rangle \bullet relayconf \\ ps' &= \langle 2, e, \phi, \perp, \psi\rangle \bullet relayconf \oplus \langle 1, e, \phi', \perp, \psi'\rangle \bullet hazardconf \end{aligned}$$

According to axiom PA7, we prove that $ps = ps'$, and subsequently $survconf = survconf'$. Therefore, managers $survCntrlr$ and $survCntrlr'$ are equivalent, and we can replace $surveyor_1$ with this equivalent UAV.

7. Discussion and Related Work

In [3], a taxonomy of different modeling dimensions of software self-adaptive systems is proposed. Also, [19] proposes a taxonomy that captures various dimensions of dynamic adaptation in automotive system software. Table 5 positions PobSAM in these two taxonomy partially. We have omitted the dimensions which are application-specific.

Flexibility of an approach to develop self-adaptive systems is realized by three different features including separation of concerns, computational reflection and component-based design [27]. We explain how PobSAM can address these requirements in the sequel.

PobSAM decouples the adaptation logic from the business logic described at an abstract level using policies. The proposed model permits us to control (adapt) the system behavior by enforcing (modifying) policies dynamically by loading new policies and configurations without re-coding actors and managers; thereby it leads to increasing system flexibility and scalability. Particularly, it is possible to change the configurations and policies of managers dynamically which is a major benefit for today's complex and evolving systems.

Computational reflection is the ability of a system to monitor and change its behavior subsequently. In PobSAM, the managers monitor the actor's behavior through the view layer and direct and adapt the system behavior. Policies provide us a high-level description of what we want without dealing with how to achieve it. Thus, using policies can be a suitable mechanism to determine if the goals are achievable using existing policy refinement techniques.

Furthermore, PobSAM is a compositional model in terms of actors, managers, policies and configurations. It is possible to change policies and configurations dynamically. Although we focused on behavioral adaptation in this paper, PobSAM can support structural adaptation as well by adding, replacing or removing actors and managers dynamically. This is an advantage over most existing approaches that concentrate on one adaptation type.

One of the main aspects of modeling a self-adaptive system is specifying adaptation requirements. To this end, we introduced a two phase adaptation strategy to pass the adaptation phase safely. Upon receiving an adaptation event by a manager, it switches to the adaptation mode. When the system reaches a safe state, the adaptation is completed by evolving the manager to the new configuration. [39] is a relevant work which extends LTL with an "adapt" operator called A-LTL to specify adaptation requirements before, during and after adaptation [39]. The adaptation semantics of A-LTL contains three types of adaptation including one-point adaptation, guided adaptation and overlapped adaptation. While under one-point adaptation semantics, the program adapts to the target program at a certain point, in guided adaptation the source program will be restricted until the system reaches a safe state. Under overlapped semantics, the target and source programs execute simultaneously. Similar to the one-point adaptation, both loose and strict adaptation modes wait for a safe state to switch to the target configuration. We can model guided adaptation using loose adaptation model in which the manager enforces suitable policies to guide the system to reach a safe state. In contrast to [39], it is impossible to have overlapped execution of new and old configurations in PobSAM. Furthermore, adaptation in [39] leads to switching to a completely new program, but adaptation of PobSAM influences the behavior of managers directly and the actors keep running normally during adaptation.

In [24] a protocol is introduced to pass the reconfiguration phase safely, i.e., the reconfiguration leaves the modified system in a consistent state, and causes no disturbance to the unaffected part of the system. Later [38] has formalized and extended this framework in order to minimize disruption and to handle hierarchical systems. In contrast to [38, 24] which consider changes at the system structure level, our modification is performed at the behavioral level. In [4], an approach without quiescence, for safe adaptation of Paradigm models to unforeseen situation (a new Paradigm model) is introduced. To this end, another well-defined Paradigm model is constructed, specifying how to migrate safely to the new model. A special component McPal is provided to coordinate the adaptation process, and each component is extended such that it can coordinate its own migration to the unforeseen model. Process algebra is used for formal analysis, particularly, to prove that the old model and the new model are abstractions of the migration process. In [38, 24, 4], multiple components are involved in adaptation, and a supervisor controls the adaptation process using the proposed protocol. In our model, changes are local to each manager and the manager is responsible to switch to the new configuration according to its policies. In contrast to [38, 24, 4], our adaptation is a one-step change where only one component is involved. Particularly, due to the fixed structure of our system model during adaptation, the issue of *avoiding inconsistent states* (due to adding/removing components) is not relevant, while it is the main concern of [38, 24]. In contrast to [4], we specify adaptation at a higher-level of abstraction, we only modify the policies of the manager and the behavior of the rest of components remains ongoing and unaltered.

Dynamic adaptation is a very diverse area of research and different communities are concerned with this issue including autonomic computing, component-based systems, software architecture, coordination models, agent-based systems and more. Structural adaptation has been given strong attention in the research community, and formal techniques have been extensively used to model and analyze dynamic structural adaptation (see [13]). Structural adaptation (or dynamic reconfiguration) is usually modeled using graph-based approaches (e.g., [35, 28, 18]) or ADL-based approaches (e.g., [26, 29]). Fewer approaches tackle behavioral adaptation as we considered. Since, we are interested in behavioral adaptation without changing the system structure, we restrict ourselves to present related work done on formal modeling of self-adaptive systems at *the behavioral level* in addition to applying policy-based approaches for engineering of self-adaptive systems. [12] proposed an approach based on the concept of proof lattice to verify if a system is in a correct state during and after adaptation in terms of satisfying the transitional-invariants. In this approach, the behavior of system during adaptation is specified using adaptation lattice in which a node is an automata denoting the behavior of a possible intermediate program. In contrast to [12], we are not concerned about formal verification of adaptive systems during adaptation. Particularly, as mentioned above, only one component is involved in our adaptation phase, therefore our adaptation process is not as complex as adaptation in [12].

Formal modeling and verification of adaptive systems at behavioral level is a young research area [14] and only a few research groups have already focused on this topic. As part of the RAPIDware project, Zhang et al. [40] proposed a model-driven approach for developing adaptive systems. In this approach, different contexts in which an adaptive program may run are determined according to high-level requirements specified by a formalism like temporal logic. The local properties of the program in each context are described formally. Then, a state-based model of the program in each context, as well as the adaptation models for the adaptations of the program from one context to another are built. Different behavioral variants of a program are modeled as Petri Nets in [40]. Furthermore, [39] introduces a model checking approach to verify the program formally.

[32] presents 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 [2], a framework, called MARS, is proposed for model-based development of adaptive embedded systems where a model consists of a set of modules. A module may have different guarded configurations which are selected depending on the current situation of modules environment. The system is specified using Synchronous Adaptive Systems (SAS) [31] and is verified using theorem proving, model checking and specialized verification methods.

The main difference of our approach with the RAPIDware project and the MARS framework is that they are mainly concerned with modeling the adaptive system at the behavioral level (even though the adaptation type may be structural), we are interested in flexible formal models to develop and model adaptive systems which adapt the system behavior by changing the system behavior rather than structure. Both RAPIDware and MARS decouple adaptation concerns from the business logic. PobSAM decouples behavioral choices in addition to adaptation logic. While in these works, the system is described at a low-level of abstraction using a semantic-level state-based formalism, we use high-level policies to control the system behavior and provide a high-level language to specify policies formally. Moreover, configurations in addition to adaptation logic are fixed in RAPIDware and MARS, while we can change configurations and adaptation policies. The ability to change configurations and adaptation logic is vital for an approach to model evolving adaptive systems.

Another close area of research is coordination models in which the interaction of objects can be controlled to achieve adaptation. While coordination models are aimed at decoupling interactions from computation and controlling interactions, PobSAM is concerned with controlling objects through controlling their behavior and decouples *the behavioral choices* and adaptation concerns from the computational environment. ARC (Actor-Role-Coordinator) [30] and PAGODA (Policy And GOal based Distributed Architecture) [36, 37] are two similar actor-based coordination models in which meta-actors control interactions of actors. ARC controls object interactions by manipulating message delivery including rerouting and reordering messages. In PAGODA, each coordinator is provided with a set of policies to coordinate actors where a simple policy may reorder messages, serialize requests and maintain a history of events. Reo [7] is another coordination model in which a graph transformation approach is used for dynamic adaptation of Reo models [23].

Employing policies as a paradigm to adapt self-adaptive systems has been given considerable attention during recent years. Policies are high-level goals describing the user requirements which can be defined and modified dynamically. In [5, 16, 17, 20, 15, 6] policies are used as the adaptation logic for structural adaptation, while we use policies as a mechanism for behavioral adaptation. Furthermore, [15] uses policies for a simple type of behavioral adaptation named parameterization, too. [25] proposes an adaptive architecture for management of differentiated networks which performs adaptation by enabling/disabling a policy from a set of predefined QoS policies. [6] presents a policy definition language for autonomic computing systems in which the policies themselves can be modified dynamically to match environmental conditions. However, this work does not deal with modeling system and it is limited to proposing an informal policy language.

8. Conclusions and Future Work

We proposed PobSAM as a formal model to develop evolving self-adaptive systems which uses policies as the main mechanism to govern and adapt the system behavior. To this end, we model a system as the composition of a set of actors: managed actors that are dedicated to the computational layer of a system and autonomous managers that coordinate actors to achieve the predefined goals using policies. This model integrates two formal methods including the algebra CA and an actor-based model to specify a system. We presented the operational semantics of PobSAM by means of labeled transition systems. We presented behavioral equivalence of CA sub-algebras including CA^a, CA^g, CA^c and CA^p according to the notions of splitting bisimilarity and prioritized splitting bisimilarity.

There is much more research to pursue in the area of modeling and verification of self-adaptive systems. In this paper, we focused on formal modeling of self-adaptive systems. Verification of different properties of adaptation and computational layers of PobSAM models is an ongoing work. As our model can support both behavioral and structural adaptations, our future research will concentrate on specifying structural adaptations.

References

[1] Autonomic computing. IBM Systems Journal, 42, 2003.

[2] Rasmus Adler, Ina Schaefer, Tobias Schule, and Eric Vecchie. From model-based design to formal verification of adaptive embedded systems. In Proceedings of the formal engineering methods 9th international conference on Formal methods and software engineering, ICFEM07, pages 7695, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] Jesper Andersson, Rogerio Lemos, Sam Malek, and Danny Weyns. In Betty H. Cheng, Rogerio Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, Software Engineering for Self-Adaptive Systems, chapter Modeling Dimensions of Self-Adaptive Software Systems, pages 2747. Springer-Verlag, Berlin, Heidelberg, 2009.

[4] Suzana Andova, Luuk Groenewegen, J. Stafleu, and Erik P. de Vink. Formalizing adaptation on-the-fly. Electr. Notes Theor. Comput. Sci., pages 2344, 2009.

[5] Richard Anthony and Cecilia Ekelin. Policy-driven self-management for an automotive middleware. In Hot Topics in Autonomic Computing on Hot Topics in Autonomic Computing, pages 5564, Berkeley, CA, USA, 2007. USENIX Association.

[6] Richard John Anthony. Generic support for policy-based self-adaptive systems. In Proceedings of the 17th International Conference on Database and Expert Systems Applications, pages 108113, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Farhad Arbab. Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science, 14(3):329366, 2004.

[8] J. C. M. Baeten, T. Basten, and M. A. Reniers. Process Algebra: Equational Theories of Communicating Processes (Cambridge Tracts in Theoretical Computer Science). Cambridge University Press, 1st edition, December 2009.

[9] Jan A. Bergstra and Jan Willem Klop. Verification of an alternating bit protocol by means of process algebra. In Mathematical Methods of Specification and Synthesis of Software Systems, pages 923, 1985.

[10] Jan A. Bergstra and C. A. Middelburg. Preferential choice and coordination conditions. J. Log. Algebr. Program., 70(2):172200, 2007.

[11] Jan A. Bergstra and C.A. (Kees) Middelburg. Model theory for process algebra. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel de Vrijer, editors, Processes, Terms and Cycles: Steps on the Road to Infinity, volume 3838 of Lecture Notes in Computer Science, pages 445495. Springer Berlin / Heidelberg, 2005.

[12] Karun N. Biyani and Sandeep S. Kulkarni. Assurance of dynamic adaptation in distributed systems. Journal of Parallel and Distributed Computing, 68(8):1097 1112, 2008.

[13] Jeremy S. Bradbury, James R. Cordy, Jurgen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In Proceedings of 1st ACM SIGSOFT Workshop on Self-managed Systems, pages 2833. ACM, 2004.

[14] Betty H. C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Software engineering for self-adaptive systems. pages 126, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In Proceedings of Int. Conf. on Distributed Applications and Interoperable Systems (DAIS), pages 114, 2003.

[16] Christos Efstratiou, Keith Cheverst, Nigel Davies, and Adrian Friday. An architecture for the effective support of adaptive context-aware applications. In Proceedings of the Second International Conference on Mobile Data Management, MDM 01, pages 1526, London, UK, 2001. Springer-Verlag.

[17] Christos Efstratiou, Adrian Friday, Nigel Davies, and Keith Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In Proceedings of IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, pages 1324, 2002.

[18] Hartmut Ehrig, Claudia Ermel, Olga Runge, Antonio Bucchiarone, and Patrizio Pelliccione. Formal analysis and verification of self-healing systems. In Fundamental Approaches to Software Engineering, pages 139153, 2010.

[19] Serena Fritsch, Aline Senart, Douglas C. Schmidt, and Siobhan Clarke. Time-bounded adaptation for automotive system software. In Proceedings of the 30th international conference on Software engineering, ICSE 08, pages 571580, New York, NY, USA, 2008. ACM.

[20] Phil Greenwood and Lynne Blair. A framework for policy driven auto adaptive systems using dynamic framed aspects. In Transactions on Aspect-Oriented Software Development II, volume 4242 of LNCS, pages 3065. Springer Berlin/Heidelberg, 2006.

[21] Christine Hofmeister. Dynamic Reconfiguration. PhD thesis, Computer Science Department, University of Maryland, 1993.

[22] Narges Khakpour, Saeed Jalili, Carolyn L. Talcott, Marjan Sirjani, and Mohammad Reza Mousavi. Pobsam: Policy-based managing of actors in self-adaptive systems. Electr. Notes Theor. Comput. Sci., 263:129143, 2010.

[23] Christian Koehler, Alexander Lazovik, and Farhad Arbab. Connector rewriting with high-level replacement systems. Electr. Notes Theor. Comput. Sci., 194(4):7792, 2008.

[24] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering, 16:12931306, 1990.

[25] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. An adaptive policy-based framework for network services management. J. Network Syst. Manage., 11(3):277303, 2003.

[26] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1996.

[27] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A Taxonomy of Compositional Adaptation. Technical report, Michigan State University Technical Report MSU-CSE-04-17, 2004.

[28] Daniel Le Metayer. Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering, 24(7):521 533, 1998.

[29] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In Proceedings of the 20th international conference on Software engineering, ICSE 98, pages 177186, Washington, DC, USA, 1998. IEEE Computer Society.

[30] Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot, and Limin Shen. Actors, roles and coordinators a coordination model for open distributed and embedded systems. In Coordination Models and Languages, volume 4038 of Lecture Notes in Computer Science, pages 247 265. Springer Berlin/Heidelberg, 2006.

[31] Ina Schaefer and Arnd Poetzsch-Heffter. Using abstraction in modular verification of synchronous adaptive systems. In Proceedings of Workshop on Trustworthy Software, Saarbrcken, Germany, 2006.

[32] Klaus Schneider, Tobias Schuele, and Marion Trapp. Verifying the adaptation behavior of embedded systems. In Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, SEAMS 06, pages 1622, New York, NY, USA, 2006. ACM.

[33] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using rebeca. Fundam. Inform., 63(4):385410, 2004.

[34] Morris Sloman and Emil C. Lupu. Engineering policy-based ubiquitous systems. Comput. J., 53(7):11131127, 2010.

[35] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In Hartmut Ehrig, Gregor Engels, Hans-Jrg Kreowski, and Grzegorz Rozenberg, editors, Theory and Application of Graph Transformations, volume 1764 of Lecture Notes in Computer Science, pages 179193. Springer Berlin / Heidelberg, 2000.

[36] Carolyn L. Talcott. Coordination models based on a formal model of distributed object reflection. Electr. Notes Theor. Comput. Sci., 150:143157, March 2006. Proceedings of the First International Workshop on Methods

and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2005).

[37] Carolyn L. Talcott. Policy-based coordination in pagoda: A case study. Electr. Notes Theor. Comput. Sci., 181:97112, 2007.

[38] Michel Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In 2nd International Workshop on Software Engineering for Parallel and Distributed Systems, pages 243 254, may 1997.

[39] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. ACM SIGSOFT Software Engineering Notes, 30(4):17, 2005.

[40] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In Proceedings of the 28th international conference on Software engineering, ICSE 06, pages 371380, New York, NY, USA, 2006. ACM.

[41] Ji Zhang, Heather Goldsby, and Betty H. C. Cheng. Modular verification of dynamically adaptive systems. In Proceedings of the 8th ACM international conference on Aspect-oriented software development, pages 161172, 2009.

Appendix A

Proof of Proposition 1

Proof. Let $\tilde{x} = \langle S_x, \to_x, \to_{x\,\surd}, {s_x}^0\rangle \in \mathbb{PCTS}$ for $x = p, p', q, q'$. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be prioritized splitting bisimulations witnessing $\tilde{p} \underline{\Leftrightarrow}_p \tilde{p}'$ and $\tilde{q} \underline{\Leftrightarrow}_p \tilde{q}'$, respectively. Then, we construct $\mathcal{R}_{\oplus} = (\{(s^0, s'^0)\} \cup \mu(\mathcal{R}_1) \cup \mu(\mathcal{R}_2)) \cap (S \times S')$ where $S = S_p \cup S_{p'}$, $S' = S_q \cup S_{q'}$, s^0 is initial state of $\tilde{p} \,\tilde{\oplus}\, \tilde{q}$, s'^0 is initial state of $\tilde{p}' \,\tilde{\oplus}\, \tilde{q}'$, and $\mu(\mathcal{R}_i) = \{(\mu_i(s), \mu_i(s')) | (s, s') \in \mathcal{R}_i\}$, $i = 1, 2$. Moreover, $\mu_1(s) = (s, \emptyset)$ and $\mu_2(s) = (s, \{\emptyset\})$.

Proof of Theorem 1

Proof. According to the definition of $\aleph$, proof of CA^P soundness is straightforward. For instance to prove soundness of axiom PA6, let $\hat{p}_1$, $\hat{p}_2$ and $\hat{p}_3$ denote the models of $p_1 = \langle o, e, \psi, \lambda, \phi\rangle \bullet c$, $p_2 = \langle o, e, \psi', \lambda, \phi\rangle \bullet c$ and $p_3 = \langle o, e, \psi \vee \psi', \lambda, \phi\rangle \bullet c$, respectively. We should prove that $\tilde{p_1} \,\tilde{\oplus}\, \tilde{p_2} = \tilde{p_3}$. Let $\mu_i^{[12]} : S_i \to S_1 \uplus S_2$ where S_i denote the state set of $\hat{p}_i$. We construct the bisimulation relation $\mathcal{R}_{\text{PA6}}$ witnessing $\hat{p}_1 \hat{\oplus} \hat{p}_2 \underline{\Leftrightarrow}_p \hat{p}_3$ as follows, where s^0 and s'^0 denote the initial states of $\hat{p}_1 \hat{\oplus} \hat{p}_2$ and $\hat{p}_3$, respectively :

$$\begin{aligned} \mathcal{R}_{\text{PA6}} &= \{(s^0, s'^0)\} \cup \{(\mu_1^{[12]}(s), s) | s \in S_3 \wedge s \neq s'^0\} \\ &\cup \{(\mu_2^{[12]}(s), s) | s \in S_3 \wedge s \neq s'^0\} \end{aligned}$$

We have the following equations according to the definition of $\aleph$,

$$[\hat{p_1} \mathbin{\hat{\oplus}} \hat{p_2}]_{\Leftrightarrow_p} = [\hat{p_1}]_{\Leftrightarrow_p} \mathbin{\tilde{\oplus}} [\hat{p_2}]_{\Leftrightarrow_p} = \tilde{p_1} \mathbin{\tilde{\oplus}} \tilde{p_2} \tag{3}$$

$$[\hat{p_3}]_{\Leftrightarrow_p} = \tilde{p_3} \tag{4}$$

and

$$[\hat{p_1} \mathbin{\hat{\oplus}} \hat{p_2}]_{\Leftrightarrow_p} = [\hat{p_3}]_{\Leftrightarrow_p} \overset{(3),(4)}{\Rightarrow} \tilde{p_1} \mathbin{\tilde{\oplus}} \tilde{p_2} = \tilde{p_3} \tag{5}$$

Proof of Theorem 2

Proof. Let $T = \langle S, \rightarrow, \rightarrow_{\surd}, s^0 \rangle \in \mathbb{PCTS}$ and $T' = \langle S', \rightarrow', \rightarrow'_{\surd}, s'^0 \rangle \in \mathbb{PCTS}$ denote two arbitrary prioritized conditional state transition systems where $\aleph \models T, T'$. Let $trm(s)$ be a function which gives a CA^p term to state $s \in S$. In addition, $trm(T) = trm(s^0)$. We define $trm(s)$ as follows:

$$trm(s) = \begin{cases} \bigoplus_j p_j & s = s^0, s \xrightarrow{\mu} s_j \wedge trm(s_j) = p_j \\ \langle o, e, \psi, \top, \phi \rangle \bullet c & s \xrightarrow{\langle \phi, switch(c), o+1 \rangle} \surd \wedge s^0 \xrightarrow{\langle \psi, tostrict(e), o+1 \rangle} s \\ \langle o, e, \psi, \bot, \phi \rangle \bullet c & s \xrightarrow{\langle \phi, switch(c), o+1 \rangle} \surd \wedge s^0 \xrightarrow{\langle \psi, toloose(e), o+1 \rangle} s \end{cases}$$

We use the following one-step reduction techniques to reduce a prioritized conditional state transition system:

- sharing of double states
- replacing $s \xrightarrow{\langle \phi, \alpha, n \rangle} s'$ and $s \xrightarrow{\langle \phi', \alpha, n \rangle} s'$ by $s \xrightarrow{\langle \phi \vee \phi', \alpha, n \rangle} s'$
- replacing $s \xrightarrow{\langle \phi, \alpha, n \rangle} \surd$ and $s \xrightarrow{\langle \phi', \alpha, n \rangle} \surd$ by $s \xrightarrow{\langle \phi \vee \phi', \alpha, n \rangle} \surd$
- removing s' and its descendant where $s \xrightarrow{\langle \phi, \alpha, n \rangle} s'$, $s \xrightarrow{\langle \phi', \alpha', n' \rangle} s''$, $n < n'$ and $\phi \Rightarrow \phi'$
- removing transition $s \xrightarrow{\langle \phi, \alpha, n \rangle} \surd$ where there are transitions $s \xrightarrow{\langle \phi', \alpha', n' \rangle} s''$ or $s \xrightarrow{\langle \phi', \alpha', n' \rangle} \surd$ where $n < n'$ and $\phi \Rightarrow \phi'$.

We say $T \rightarrowtail T'$ when T' is obtained using the introduced reductions from T and write $\twoheadrightarrow$ for transitive and reflexive closure of $\rightarrowtail$. We say T is in normal form, if it can not be reduced by reduction rules anymore. It can be proved that

(1) for all $T, T' \in \mathbb{PCTS}, T \twoheadrightarrow T'$ implies $T \mathrel{\underline{\Leftrightarrow}_p} T'$.

(2) for all $T, T' \in \mathbb{PCTS}$ that are in normal form, $T \mathrel{\underline{\Leftrightarrow}_p} T'$ implies $T = T'$.

(3) $T \rightarrowtail T'$ implies CA$^p \vdash trm(T) = trm(T')$.

Let $pcts(p)$ and $pcts(p')$ indicate the models of p and p' in $\aleph$ where $pcts(p) \mathrel{\underline{\Leftrightarrow}_p} pcts(p')$. We reduce $pcts(p)$ and $pcts(p')$ to their normal form indicated by T_p and $T_{p'}$. Thus, we conclude from property (1) that

$$pcts(p) \mathrel{\underline{\Leftrightarrow}_p} T_p \tag{6}$$

$$pcts(p') \mathrel{\underline{\Leftrightarrow}_p} T_{p'} \tag{7}$$

From the fact $pcts(p) \mathrel{\underline{\Leftrightarrow}_p} pcts(p')$, (6) and (7), we conclude $T_p \mathrel{\underline{\Leftrightarrow}_p} T_{p'}$. Since T_p and $T_{p'}$ are in normal form, we prove $T_p = T_{p'}$ according to property (2).

It is easy to show that $\mathrm{CA}^p \vdash trm(pcts(p)) = p$ and $\mathrm{CA}^p \vdash trm(pcts(p')) = p'$. It follows from (3) that $\mathrm{CA}^p \vdash trm(T_p) = p$ and $\mathrm{CA}^p \vdash trm(T'_p) = p'$. Therefore, from the fact $T_p = T_{p'}$, we conclude $\mathrm{CA}^p \vdash (trm(T_p) = trm(T_{p'}))$ and $\mathrm{CA}^p \vdash p = p'$.

Appendix B

Proof of Theorem 3

Proof. To prove left to right, let $\mathcal{R}_c$ be the prioritized splitting bisimulation witnessing $c_1 \mathrel{\underline{\Leftrightarrow}_p} c_2$. Then we construct $\mathcal{F}_g$ and $\mathcal{R}_p$ as formulas 8 and 9 witnessing $\Psi(g) \mathrel{\underline{\Leftrightarrow}_p} \Psi(g')$ and $p \mathrel{\underline{\Leftrightarrow}_p} p'$, respectively. Moreover, $f(s,n)$ is a function which gives the nth component of a triple, e.g., $f((s', s'', s'''), 2) = s''$.

$$\mathcal{F}_g = \{(f(s_1,2), f(s'_1,2)) \mid (s_1, s'_1) \in \mathcal{R}_c \wedge (\exists_{s_2}.s_1 \xrightarrow{\langle\phi,\alpha,1\rangle} s_2 \vee s_1 \xrightarrow{\langle\phi,\alpha,1\rangle} \surd)\} \tag{8}$$

$$\mathcal{R}_p = \{(f(s_1,3), f(s'_1,3)) \mid (s_1, s'_1) \in \mathcal{R}_c \wedge (\exists_{s_2}.s_1 \xrightarrow{\langle\phi,\alpha,n\rangle} s_2 \vee s_1 \xrightarrow{\langle\phi,\alpha,n\rangle} \surd) \wedge n \neq 1\} \tag{9}$$

To prove right to left, let $\mathcal{R}_p$ and $\mathcal{F}_g$ be the prioritized splitting bisimulations witnessing $p \mathrel{\underline{\Leftrightarrow}_p} p'$ and $\Psi(g) \mathrel{\underline{\Leftrightarrow}_p} \Psi(g')$, respectively. Then, formula 10 is a relation witnessing $c \mathrel{\underline{\Leftrightarrow}_p} c'$ where T_{c_1} and T_{c_2} indicate the models of c_1 and c_2, respectively .

$$\begin{aligned} \mathcal{R}_c = \{((t, s_g, s_a), (t, s'_g, s'_a)) | (s_a, s'_a) \in \mathcal{R}_p \wedge (s_g, s'_g) \in \mathcal{F}_g \\ \wedge (t, s_g, s_a) \in T_{c_1} \wedge (t, s'_g, s'_a) \in T_{c_2}\} \end{aligned} \tag{10}$$

Appendix C

Proof of Theorem 4

Proof. To prove left to right, let $\mathcal{R}_m$ be the prioritized splitting bisimulation witnessing $m \mathrel{\underline{\Leftrightarrow}_p} m'$. We construct bisimulation relation $\mathcal{R}_{init}$ witnessing $c_{init} \mathrel{\underline{\Leftrightarrow}_p} c'_{init}$ as formulas (11), where $S_{c_{init}}$ and $S_{c'_{init}}$ denote the state set of c_{init} and c'_{init}, respectively. Therefore, we prove $c_{init} = c'_{init}$ according to Theorem 3.

$$\mathcal{R}_{init} = \mathcal{R}_m \cap (S_{c_{init}} \times S_{c'_{init}}) \tag{11}$$

To prove right to left, let $c_l \in C$ and $c'_k \in C'$ be two prioritized splitting bisimilar configurations. If there are transitions $(s_1, swicth(c_i), s_2) \in \rightarrow_{c_l}$ and $(s'_1, swicth(c'_j), s'_2) \in \rightarrow_{c'_k}$ where $(s_1, s'_1) \in \mathcal{R}$ and $(s_2, s'_2) \in \mathcal{R}$, then we have $c_i = c'_j$. Then, we can construct a relation $\mathcal{R}_{ij}$ witnessing $c_i \underline{\Leftrightarrow}_p c'_j$ according to Theorem 3. According to Definition 7 and Theorem 3 we have $c_{init} \underline{\Leftrightarrow}_p c'_{init}$. Starting from $c_{init} \underline{\Leftrightarrow}_p c'_{init}$, we construct all $\mathcal{R}_{ij}$ for $c_i \in C$ and $c'_j \in C'$ where $c_i = c'_j$. The formula 12 is a relation witnessing $m \underline{\Leftrightarrow}_p m'$.

$$\mathcal{R}_m = \bigcup_{c_i = c'_j} \mathcal{R}_{ij} \tag{12}$$