MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems

DIDAC GIL DE LA IGLESIA, Linnaeus University
DANNY WEYNS, Linnaeus University

Designing software systems that have to deal with dynamic operating conditions, such as changing availability of resources and faults that are difficult to predict, is complex. A promising approach to handle such dynamics is self-adaptation that can be realized by a MAPE-K feedback loop (Monitor-Analyze-Plan-Execute plus Knowledge). To provide evidence that the system goals are satisfied, given the changing conditions, the state of the art advocates the use of formal methods. However, little research has been done on consolidating design knowledge of self-adaptive systems. To support designers, this paper contributes with a set of formally specified MAPE-K templates that encode design expertise for a family of self-adaptive systems. The templates comprise: (1) behavior specification templates for modeling the different components of a MAPE-K feedback loop (based on networks of timed automata), and (2) property specification templates that support verification of the correctness of the adaptation behaviors (based on timed computation tree logic). To demonstrate the reusability of the formal templates, we performed four case studies in which final-year Masters students used the templates to design different self-adaptive systems.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Formal methods

General Terms: Design, Verification

Additional Key Words and Phrases: Formal templates, MAPE-K, self-adaptation

ACM Reference Format:
Didac Gil de la Iglesia, Danny Weyns, 2015. MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems. *ACM Trans. Autonom. Adapt. Syst.* X, X, Article X (January 2015), 28 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Designing contemporary software intensive systems, such as mobile applications, multi-robot systems, and networked smart homes, and guaranteeing the system goals is complex due to the dynamic conditions in which these systems operate. Examples of dynamics are changing availability of resources and faults that are difficult to predict. This situation has led to the development of self-adaptive systems, i.e., systems that adapt to dynamics by reconfiguring their structure and modify their behavior at run-time with little or no human intervention [Cheng et al. 2009; de Lemos et al. 2013; Dobson et al. 2006]. The underlying principle of self-adaptation is separation of concerns [Weyns et al. 2013b]. A self-adaptive system typically consists of a managed system and a feedback loop. The managed system deals with the domain concerns for the users, while the feedback loop deals with adaptation concerns of the managed system (e.g., optimize the managed system for different operating conditions, heal the managed system when a fault is detected, etc.). In this research, we focus on architecture-based self-adaptation [Oreizy et al. 1998; Garlan et al. 2004; Kramer and Magee 2007; Weyns et al. 2012b]. In architecture-based self-adaptation, the system reasons about a model of itself and its environment and adapts when needed according to some adaptation goals. The feedback loop typically consists of Monitor, Analyze, Plan, and Execute com-

ponents, complemented with runtime models (Knowledge) of the managed system, its environment and adaptation goals. This structure is referred to as MAPE-K [Kephart and Chess 2003].

One important challenge in engineering self-adaptive systems is to provide evidence that the system goals are satisfied [Cheng et al. 2009; de Lemos et al. 2013; de Lemos et al. 2014]. To that end, the state of the art in architecture-based self-adaptation advocates the use of formal methods. A recent systematic literature survey [Weyns et al. 2012a] identified a total of 75 papers that use formal methods in self-adaptive systems. The study revealed that designers usually specify models and required properties of the self-adaptive system for the problem at hand from scratch. There is little systematic consolidation of the design knowledge for future use. [Filieri et al. 2012] points out that such consolidation is an important research challenge for self-adaptive systems.

During the past years, we have been studying self-adaptation in a number of application domains, with a particular focus on robustness and openness requirements. Some of the recent applications we studied are an intelligent monitoring system to detect traffic jams [Iftikhar and Weyns 2012], a mobile system to support outdoor learning activities [Gil de la Iglesia and Weyns 2013], a robotic transportation system [Iftikhar and Weyns 2014a; 2014b] and a mobile digital storytelling application [Weyns et al. 2014]. In the course of designing and building these self-adaptive applications, we derived a set of *MAPE-K Formal Templates* for designing feedback loops of self-adaptive systems. These templates encode design expertise that can help designers with specifying models and properties for new similar self-adaptive software systems. In this paper, we document these templates and evaluate their reusability. Concretely, the contribution of this paper is twofold. First, we document a set of MAPE-K Formal templates that comprise: (1) behavior specification templates for modeling the different components of a MAPE-K feedback loop and their interactions, and (2) property specification templates for specifying required properties of the adaptation behaviors enabling formal verification of the behaviors. Second, we empirically evaluate the reusability of the Formal Templates in four cases studies in which final-year Master students in Software Engineering applied the templates to design different self-adaptive applications.

In this research, we use networks of timed automata (TA) as the formal language for the specification of behavior specification templates, and timed computational tree logic (TCTL) for the property specification templates. The survey [Weyns et al. 2012a] identified automata as one of the mostly used languages in the field. TA are automata extended with clock variables that comprise behavior states and transitions between states that represent actions. Clock variables are used to synchronize behaviors, together with channels that enable communication between behaviors. A network of TA allows specifying the behaviors of MAPE-K components that synchronize through clock variables and interact via channels, which models the behavior of the feedback loop of a self-adaptive system.

TCTL is a formal language based on computational tree logic extended with clock variables that allows the specification of properties. Expressions in TCTL describe state and path formulae enabling the verification of properties of interest, such as reachability (a system should/can/cannot etc. reach a particular state or states), liveness (something eventually will hold), safety (given a particular condition in the system, all possible system executions should hold certain desired conditions or avoid specific undesired conditions), etc. TA and TCTL can be combined to specify behaviors of self-adaptive systems and to verify that the desired properties hold.

The remainder of the paper is structured as follows. In Section 2, we discuss related work. Section 3 elaborates on the target domain that we focus on. This section describes two example self-adaptive applications from which the MAPE-K Formal Templates were derived. Section 4 presents the behavioral specification templates for modeling feedback loops of self-adaptive systems. In Section 5, we present the property specification templates that support the verification of self-adaptation behaviors. We illustrate the MAPE-K Formal Templates with excerpts of two self-adaptive applications introduced in Section 3 . In Section 6, we report the results of the four case studies that demonstrate the reusability of the MAPE-K Formal Templates. Finally, we draw conclusions in Section 7.

2. RELATED WORK

We start the discussion of related work with a selection of representative approaches that use formal approaches for the design of self-adaptive systems. Then we zoom in on studies that present patterns for self-adaptive systems. We conclude with a selection of studies on property specification templates presented in software engineering in general.

2.1. Formal Approaches to Self-Adaptation

Providing evidence that self-adaptation properties are satisfied is subject of active research [Magee and Maibaum 2006; Vassev and Hinchey 2009; Tamura et al. 2012]. During the recent Dagstuhl seminar *Software Engineering for Self-Adaptive Systems: Assurances* [de Lemos et al. 2014], several approaches for assurances of self-adaptive systems were studied. Some approaches use formal methods at design time (e.g., model checking), others at runtime or a combination of both. Our research focusses on providing assurances for system requirements during system design.

The process presented in [Zhang and Cheng 2006] aims to create formal models for adaptive systems, verify the models and automatically translate the models into executable programs. The approach guarantees conformance between the models and programs using model-based testing. In follow up work [Zhang et al. 2009], the authors model a dynamically adaptive program as a collection of steady-state programs and a set of adaptations that realize transitions among steady state programs in response to environmental changes. To handle state explosion, the authors propose a modular model checking approach.

FOrmal Reference Model for Self-adaptation (FORMS), a comprehensive reference model for self-adaptive systems [Weyns et al. 2012b], builds on established principles of self-adaptation, such as architecture-based self-adaptation, computational reflection, and MAPE-K. The reference model offers a vocabulary of a small number of primitives and a set of relationships among them that delineates the rules of composition. The model is formally specified, which enables engineers to precisely define the key characteristics of self-adaptive software systems, and reason about them.

POISED [Esfahani et al. 2011] is a quantitative approach that focusses on adaptation under uncertainty. The approach builds on possibility theory (grounded in fuzzy mathematics) to assess both the positive and negative consequences of uncertainty. In POISED, adaptation decisions are used to reconfigure customizable software components to improve the system's quality of service.

The MechatronicUML proposed in [Giese and Schäfer 2013] aims to support model-driven development and verification of safety guarantees of embedded real-time self-adaptive systems. Mechatronic UML modeling is based on a rigorously specified subset of UML. This approach offers verification at design time to provide assurances regarding the system functional and non-functional requirements, by describing the complete system behavior.

This sample of studies of the past decade underpin the relevance of rigorous modeling and verification in the design of self-adaptive systems. However, they also show that there is a lack of consolidating expertise in the design of self-adaptive systems' behaviors, allowing engineers to benefit from systematic knowledge [Shaw and Clements 2006; Weyns et al. 2012a]. This paper contributes with a set of templates for the specifications of MAPE-K behaviors that consolidate design expertise offering support to designers of a family of self-adaptive systems.

2.2. Design Patterns

A common approach to consolidate design knowledge is by means of documenting design patterns. In the field of architecture-based self-adaptive systems, only a few patterns have been documented.

Gomaa and Huseein [2004] present reconfiguration patterns for dynamic evolution of software architectures that are based on sequences of adaptation steps (e.g., stopping/starting, (un)linking, adding/removing). In [Gomaa et al. 2010], the authors employ reconfiguration patterns in the context of self-managed service-oriented software systems, while [Esfahani and Malek 2010] show their utility in the design of architecture-based middleware solutions.

Edwards et al. [2009] propose an adaptive software architecture style, consisting of: (1) a basic bottom layer with application components that control a robot, and (2) one or more meta layers with meta-level components that implement fault-tolerance, dynamic update, resource discovery, redeployment, etc. Each layer may adapt the layer beneath. The authors use the approach for the design and implementation of self-adaptive behavior in robotics software.

Ramirez and Cheng [2010] document a set of 12 patterns derived from a number of studies in self-adaptive systems. Each pattern is documented using its intent, context, structure using UML class diagrams, and consequences, among other items. The presented patterns focus at the software design level. They aim to facilitate the design and construction of self-adaptive systems by providing alternative solutions for adaptation in the systems' implementations.

In a recent study [Weyns et al. 2013], the authors present a set of five patterns for decentralized control in self-adaptive systems that were derived from different studies. The patterns focus on coordination and interaction of MAPE-K feedback loops, and include a master-slave MAPE pattern, coordination pattern, and information sharing pattern for large self-adaptive systems. The primary focus of the patterns is on structural aspects of the architectural design of self-adaptation logic. The patterns are illustrated with concrete instances from which they were derived.

The presented efforts document patterns for self-adaptive systems ranging from a high-level architecture to concrete design. The primary focus is on structural aspects. The work presented in this paper complements these efforts with templates to support engineers to rigorously specify behaviors of interacting MAPE-K components.

2.3. Property Specification Patterns

To the best of our knowledge, no related work exists on property specification patterns dedicated to self-adaptive systems. We discuss a number of general approaches that do not focus on self-adaptive systems and then zoom in on initial ideas to define property specification patterns for self-adaptive systems.

Dwyer et al. [1998] introduce property specifications templates for the verification of finite-state machines. The templates allow the expression of recurring properties in a generalized form to rigorously verify system requirements. Other examples of property specifications have been proposed for real-time systems [Koymans 1990] and systems with probabilistic quality requirements [Grunske 2008]. Bianculli et al. [2012] present an extensive analysis of the usage of documented specification properties for a large body of specifications for service-based applications of a banking company. Existing property specification patterns are not specified to be applied in self-adaptive systems, and recent efforts in which properties for self-adaptive systems have been specified are not consolidated to support designers of other systems.

As model checking is one of the prominent approaches to verify system properties, we analyzed existing pieces of work that employ model checking techniques in self-adaptive systems [Weyns et al. 2012a]. From this analysis, we derived an interesting model that maps the different types of behaviors of self-adaptive systems to four zones of the state space: normal behavior, undesired behavior, adaptation behavior, and invalid behavior. Properties of interest with respect to self-adaptation typically map to transitions between different zones. For example, a property may express whether invalid states can be reached from a system's normal behavior, or whether a system adapts correctly from undesired behavior via adaptation behavior back to normal behavior. This zone model provides a potential basis for a systematic documentation of specification properties for self-adaptive systems. We refer the interested reader for additional information to [Weyns et al. 2012a].

3. TARGET DOMAIN

The formal templates we present in this paper encode knowledge we gained from the formalization of adaptive behaviors for a number of self-adaptive systems in mobile learning [Gil de la Iglesia and Weyns 2013; Gil de la Iglesia and Weyns 2013; Gil de la Iglesia 2014], traffic control [Iftikhar and Weyns 2012], robotics [Iftikhar and Weyns 2014a; 2014b] and mobile digital storytelling [Weyns

et al. 2014]. These systems share a set of characteristics that define the target domain of the templates:

— Systems comprise software deployed on distributed nodes;
— The nodes have an explicit position in the environment (and may be mobile);
— The nodes have continual communication access;
— Dynamics in the system and the environment are orders of magnitude lower than the speed of communication and the execution of the software;
— Resources in the system may come and go.

As we consider distributed self-adaptive systems, the managed system typically comprises multiple local managed systems that are deployed on different nodes and are adapted by one or more MAPE-K feedback loops. Our particular focus is on self-adaptation to deal with *robustness* and *openness* requirements. To that end, the self-adaptive system can *add* resources to the system or *remove* resources. Resources are controllable parts of the managed system, such as nodes, subsystems and components. Regarding robustness, we study self-adaptation to deal with parts of the system that fail. Regarding openness, we study self-adaptation to deal with parts that come and go dynamically.

Below we describe two example applications from the self-adaptive systems we have built and from which the templates were derived[1]. We use these applications throughout the paper to illustrate the different templates. We selected these applications for two reasons: first, they have a proper level of complexity to illustrate the use of the MAPE-K templates, and second, they require self-adaptation for complementary quality requirements.

As explained in the introduction, we use TA and TCTL for the formalization of the MAPE-K templates. Appendix A summarises the languages primitives of TA and TCTL. For the specification of models and verification of properties we used the Uppaal tool [Behrmann et al. 2006].

3.1. Mobile Learning Application

One of the mobile learning applications we studied supports pupils with learning geometry concepts using GPS-enabled mobile devices [Gil de la Iglesia and Weyns 2013]. The resources in this domain are GPS services. The activities require the pupils to work in groups since two devices are required for distance calculations, three devices for triangulation, and more devices for more advanced activities. We designed two MAPE-K feedback loops to deal with two robustness concerns. Firstly, the environmental conditions may affect the GPS reliability and consequently affect the quality of the GPS-based measurements. The first feedback loop ensures that only GPS services are used with a minimum level of accuracy in their measurements. Secondly, the number of GPS services available in a group may not match the number of services that are required. The second feedback loop ensures that a group has the required number of services to carry out the activity. The MAPE-K feedback loops are distributed among the mobile devices. In this paper, we focus on the second concern (ensuring that groups have the right number of services to carry out their task).

Fig. 1-top shows a timed automaton that specifies the behavior of the environment in which the sky can either be *Clear* or *Cloudy*. Transitions between these states are modeled using specific functions (*getNextCloudy*, *getNextClearing*, *isValid*) and conditions on a *timer*.

Fig. 1-middle specifies the part of the GPS component behavior that deals with the quality of the measured locations. The GPS service is either activated or deactivated depending on the sky conditions. These conditions are provided by the environment automaton via the *qualityGPSModule** signals. The updateGPSService() function activates or deactivates the local GPS service.

The Probe behavior shown in Fig. 1-bottom senses the state of the GPS service and notifies the relevant MAPE-K loop when needed (via the *GPSwentDown* signal).

The result of this combination of automata is that a group of phones may no longer have the right number of services to complete the task, as services are deactivated. We focus on this number of

[1]The complete set of applications from which the templates were derived are available on the project website http://homepage.lnu.se/staff/daweaa/MAPE-K-Templates.htm [Gil de la Iglesia et al. 2014]

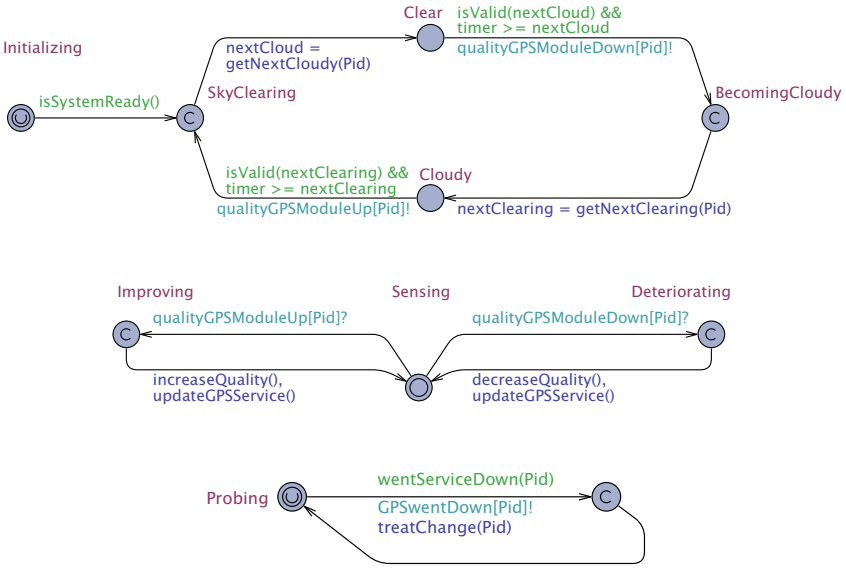

Fig. 1. Sky environment (top), GPS module (middle) and GPS-Probe behaviors for a mobile learning application

services requirement as a robustness concern, extending the managed part of the application with feedback loops to deal with the concern.

3.2. Logistical Robotic System

The second application is a robotic system in which robots have to perform transportation tasks in a warehouse following a graph-based routing layout. To drive in the warehouse, the robots maintain a representation of the routing layout. The resources in this domain are map elements such as lanes, location on the layout, etc. We focus here on a scenario in which robots are instructed to disable or enable particular elements of the layout by adding or removing parts where the robots can drive or perform tasks (e.g., remove a lane for maintenance or add a temporal drop location to deliver loads). We specified the layout that each robot uses to drive in the warehouse by an automaton. Places in this automaton represent locations of the graph-based routing layout and transitions represent movements of robots from one location to another. Fig. 2-left shows the behavior that represents the interface that allows a manager to manipulate the map elements of the layout during execution. We defined a set of actions that a manager can perform: adding (*bAdd-*) and removing (*bRem-*) locations (*-Loc*), edges[2] (*-Edge*) and destinations (*-Dest*). For instance, *bRemLoc?* in Fig. 2-left collects requests from the manager to remove a location from the layout, and notifies a robot with identity *RiD* to update its knowledge of the layout. Manipulating the layout may be constrained by the current conditions, e.g., a robot should not disable a lane it is driving on. Fig. 2-right specifies a behavior that controls the current robot location and the target destination. To deal with the manipulations of the layout we extended the domain logic of the robots with a MAPE-K feedback loop.

4. BEHAVIOR SPECIFICATION TEMPLATES FOR MODELING MAPE-K FEEDBACK LOOPS

Fig. 3 shows a high level model of a self-adaptive system where a managed system is extended with a feedback loop composed of MAPE-K components. A Monitoring component (number 2 in Fig. 3) acquires information from the managed system and its environment, and it updates the Knowledge (1) accordingly. The Analyze component (3) uses the knowledge to determine the need for adaptation of the managed system with respect to the adaptation goals. If adaptation is required,

[2]Adding or removing an edge in the map translates to enabling or disabling the related lane on the layout.

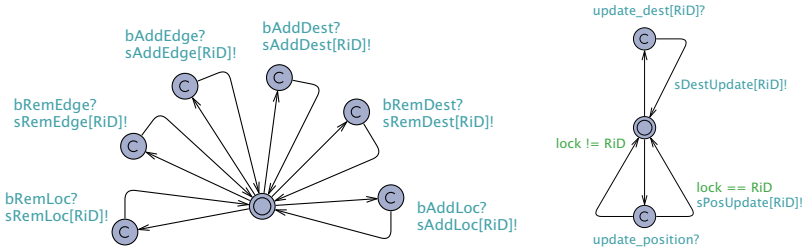

Fig. 2. Behavior specifying interfaces that managers use to update the map layout (left) and updating the robot's current location and destination (right)

the Plan component (4) puts together a plan consisting of adaptation actions that are executed by the Execute component (5) adapting the managed system as needed.

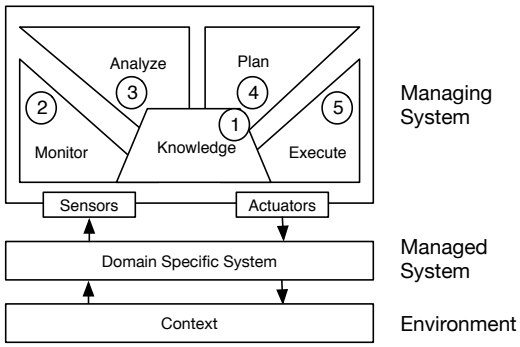

Fig. 3. MAPE-K. Model for self-adaptive systems [Kephart and Chess 2003]

We now present the set of formally specified templates for designing the behaviors of MAPE-K components. For each template, we start by presenting concrete instances of the specific MAPE component for the two application scenarios presented in Section 3. Then we specify the reusable behavior template for the the MAPE component that defines its essential elements. These elements were derived from the concrete MAPE component specifications of the different applications we built as illustrated by the concrete instances.

Due to space constraints, we cannot provide a detailed specification of all aspects of the templates, including time and event triggering mechanisms, specification of data structures, detailed specification of all functions, etc. For a complete specification and explanation of the templates, supported by different examples, we refer to the project website.

4.1. Knowledge

To perform self-adaptation, the Monitor, Analyze, Plan and Execute components use models (Knowledge) that provide an abstraction of relevant aspects of the managed system, its environment, and the self-adaptation goals. In line with FORMS [Weyns et al. 2012b], we divide knowledge in four parts (which are technically specified as four *struct* definitions in Uppaal) as follows:

— *ManagedSystemKnowledge* provides an abstraction of the managed system. This part of the knowledge represents relevant information of the resources in the managed system, such as the status of resources (used or not), resource dependencies, or quality properties.
— *EnvironmentKnowledge* provides an abstraction of the environment representing the *context* in which the self-adaptive system is situated and operates.

— *ConcernKnowledge* describes the knowledge w.r.t. the adaptation concern of interest; this part specifies the *requiredResources* to realize the goals of the self-adaptive system.
— *AdaptationKnowledge* represents runtime information that is shared between the MAPE behaviors; we distinguish between *flags* that are used for indirect synchronization of behaviors and *workingData* that represents data that behaviors use to perform their functions (e.g., historical data for analysis, plans for adaptation, etc.).

Declaration 1 shows an excerpt of the Managed System Knowledge. The abstraction allows us to define the number of nodes (*Nodes*) and resources (*NRes*) of the system, as well as a concrete resource (*Resource*) that encapsulates the state representing the type (*resourceType*) and specific quality properties (*resourceQuality*) of a resource that need to be achieved by the MAPE components. Additionally, Declaration 2 shows how the MAPE behaviors can access the knowledge (*getResourceUsed()*) and update (*setResourceUsed()*) it, in order to perform self-adaptation.

```
const int Nodes = <define>; //number of nodes in the System
const int NRes = <define>; // number of existing resources

typedef struct {
    int resourceType;
    int resourceQuality;
}Resource;

Resource resources[NRes];

struct {
    int usedResources; //State representing the number of used resources
    int availableResource[NRes]; // State representing the available nodes
    int usedNode; //State representing whether the node currently belongs to a group or not
}ManagedSystemKnowledge[Nodes];
```

Declaration 1. Specification of the ManagedSystemKnowledge

```
// Getter for a Resource module
int getResourceUsed(int node, int resource){
  return ManagedSystemKnowledge[node].availableResource[resource];
}

// Setter for a Resource module
void setResourceUsed(int node, int resource, int status){
    ManagedSystemKnowledge[node].availableResource[resource] = status;
}
```

Declaration 2. Specification for manipulating the ManagedSystemKnowledge

Declaration 3 shows an excerpt of the instantiation of ManagedSystemKnowledge for the Mobile learning case. In this example, multiple groups can be defined (*MVDGroup*), in which mobile phones (*MobilePhone*) are resources that are controlled by the MAPE components.

```
const int Phones = 12;

typedef struct {
    int status;          // Either the Phone is used=1, or Free=0
    int GPS_Quality; // –1=Broken, 0=Undesirable, 1=OK
    int prev_min_accuracy; //This variable is to control that there is a change in the GPS accuracy
                           // requirement
    ...;
}MobilePhone;

typedef struct {
    MobilePhone member[Phones];
    int nMembers;
}MVDGroup;

MVDGroup ManSysPhones[MVD];
```

Declaration 3. Specification of ManagedSystemKnowledge (Mobile learning case)

4.2. Monitor

Monitor collects data (through probes) from the managed system and possibly the environment to update the Knowledge. Before updating the Knowledge, the Monitor may preprocess the collected data. Examples of preprocessing are normalization of data, filtering and aggregating data.

4.2.1. Mobile Learning Case. We specified an automaton to describe the GPS-Monitor behavior that is responsible for updating knowledge about the membership of phones in groups.

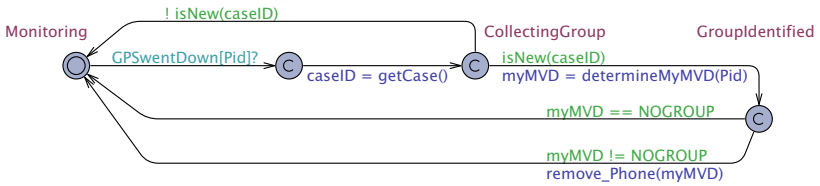

Fig. 4. Mobile learning GPS-Monitor behavior

Concretely, the Monitor behavior: (1) identifies whether GPS services go down, (2) checks whether a GPS going down was already reported, (3) identifies the group in which the mobile device of the GPS service was used, and (4) updates the knowledge with the identified group. The four steps are represented by four states of the Monitor automaton (Fig. 4). In the *Monitoring* state, the behavior is waiting to be notified by the Probe (*GPSwentDown[Pid]?*). If it is a new notification (*isNew(caseID)*), the group identification starts at *CollectingGroup* (through *determineMyMVD(Pid)*), and finally, after *GroupIdentified*, the behavior updates the knowledge of phone memberships via the *remove_Phone(myMVD)* function. The *remove_Phone* function (see Declaration 4) modifies knowledge related to the resources available in a group . Concretely, the function removes a failing phone *PhoneId* from a group *GroupID* and it decreases the number of available resources in the group.

```
void remove_Phone(MVD_id GroupId, Phone_id PhoneId){
    ManSysPhones[GroupId].member[PhoneId] = 0;
    ManSysPhones[GroupId].nMembers−−;
}
```

Declaration 4. Specification of Knowledge manipulation (Mobile learning case)

A variant of this behavior following a time-triggered approach is documented in Appendix A.

4.2.2. Robot Transportation Case. We designed a monitor behavior that is responsible for updating knowledge about: (1) desired modifications in the layout, and (2) the current position and destination of the robot. Therefore, the behavior monitors two data sources that are required by the adaptation logic. First (Fig. 5-top), the behavior monitors events that trigger changes of the layout; signals can be received via the *sAddEdge*, *sRemEdge*, *sAddLoc*, *sRemLoc*, *sAddDest* and *sRemDest* channels. Second (Fig. 5-bottom), the behavior monitors the robot's context, which is needed to determine whether an adaptation can take place or should be postponed until required conditions are satisfied (e.g., the robot is driving to a destination that needs to be removed). This is specified in the behavior through two channels that identify updates of the current robot location (*sPosUpdate*) and targeted destination (*sDestUpdate*). The monitor behavior triggers the analyze behaviour via the *analyze[RiD]!* signal.

4.2.3. Monitor Behavior Template. We now present a Monitor behavior template that consolidates our knowledge in the design of monitoring behaviors.

Triggering events, such as *GPSwentDown[Pid]?* in the mobile learning case and *sDestUpdate[Rid]?* in the robot transportation case, are generalized to a triggering mechanism that initiates data collection. Fig. 6 shows an event triggering mechanism specified with the *Monitor[Node_ID]?* signal. In the mobile learning case, this was specified through the *determineMyMVD()* function;

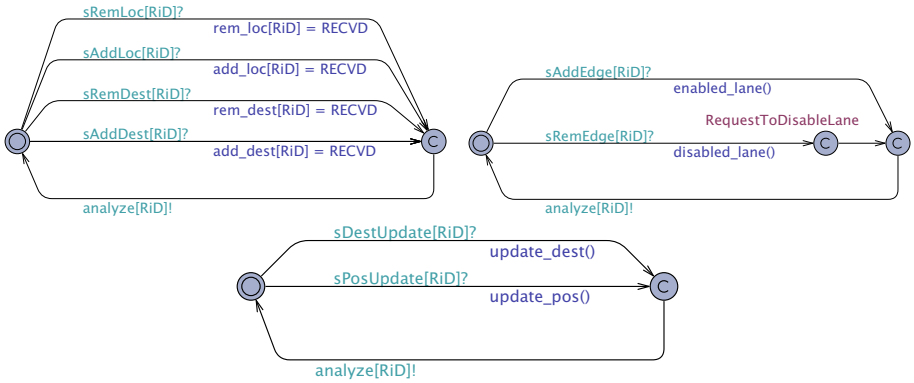

Fig. 5. Robot transportation interface Monitor behaviors

in the robot application, the data to monitor was self-contained in the signals. We generalize this step in the *getData()* function, which is responsible for collecting the data items that the behavior is expected to monitor. We include a preprocessing step in the monitor for such behaviors that may be needed for data normalization, aggregation, etc. (if *isPreprocessingRequired()* then *preprocess()*). In the mobile learning case, a preprocessing step was required to avoid handling repeated requests.

Finally, when new data items are found (*NewChange*), the monitor behavior updates the knowledge and notifies the related Analyze components. We specify this process in the Monitor behavior template with the *updateKnowledge()* function, followed by a signal that triggers the Analysis behaviour (*Analyze[Node_ID]!*). In the mobile learning case the *remove_phone()* function is a refinement of the *updateKnowledge()* function, and the analysis is time triggered. For the robot transportation case this was modeled via the different update functions (*rem_loc[Rid] = RECVD*, *update_dest()*, etc.) followed by the *analyze[Rid]!* signal.

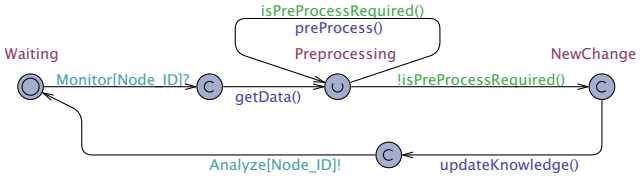

Fig. 6. Monitor behavior template with event-triggering

In summary, the Monitor behavior consists of the following steps: monitor triggering, collecting data, preprocessing data, updating working data, and signaling analyze behavior(s).

Alternatively, we specify a time-triggered template (all the template variations are documented on the project website). Fig. 7 shows a time-triggered Monitor that periodically initiates its behavior to collect sensed data via *getData()*. The monitor frequency is defined by a time condition ($t==Period$) and a state invariant ($t<=Period$).

4.3. Analyze

Analyze is responsible for determining whether adaptation actions are required based on the monitored state of the managed system, the environment and the adaptation concern of interest.

4.3.1. Mobile Learning Case. We designed an Analyze behavior that determines whether groups have enough GPS services to perform the learning activities. Fig. 8 shows the automaton of the analyze behavior that is triggered every 5 time units. The behavior uses the activity requirements

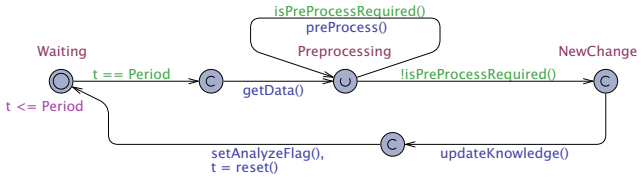

Fig. 7. Monitor behavior template with time-triggering

and the number of used GPS resources in the group to perform analysis. The *getRequired()* and *getUsed()* functions provide this knowledge by accessing ConcernKnowledge and ManagedSystemKnowledge. This knowledge is then used to evaluate whether the available GPS services in the group are sufficient to realize the activity goals (using the comparison functions). The analysis results are used to coordinate with the Plan to take action when needed (e.g., *SH_MVD_Incomplete[Mid]!*).

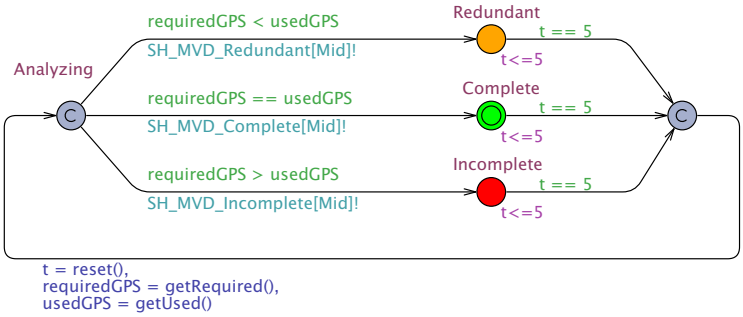

Fig. 8. Analyze behavior for the mobile learning case

4.3.2. Robot Transportation Case. The Analyze behavior needs to identify whether the representation of the layout used by the robot corresponds to the real layout defined by the administrator. We specified three functions that determine: (1) whether the current layout representation is correct (*noNeedForAdaptation()*), (2) whether new lanes, locations or other resources need to be added (*needAdaptationAdd()*), or (3) whether resources need to be removed from the layout (*needAdaptationRemove()*). Fig. 9 shows the event-triggered analyze behavior for the robot scenario.

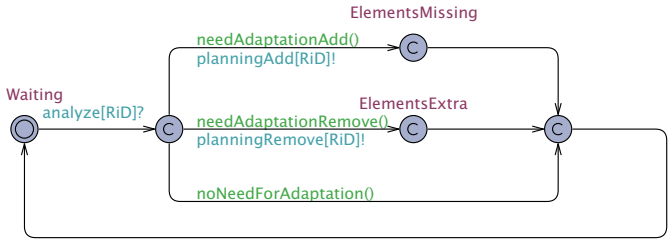

Fig. 9. Analyze behavior for a robotic case

The behavior can take three different transitions from the *Waiting* state based on the results of the functions. A transition via *ElementsMissing* is taken in case map elements need to be enabled or added. A transition via *ElementsExtra* is taken when one (or more) elements in the map need to be disabled or removed. In both cases, the Analyze behavior signals the Plan behavior of the same

robot (*planning*[RiD]!*). In the remaining case, the Analyze behavior directly returns to the *Waiting* state and waits until a new analysis is required.

4.3.3. Analyze Behavior Template. Now, we present the Analyze behavior template (Fig. 10) that generalizes the analyze behaviors from different applications, as illustrated with the examples above.

The Analyze behavior compares the required resources with the resources in use, taking into account the current context and working data (such as GPS services or lanes in a robot layout). Functions such as comparing quantities of GPS services and checking that the robot works with an updated layout (*needAdaptationAdd()*) are generalized to a *matchResources()* function. We provide an example (*needAdaptationAdd*) in Declaration 5. To support the analysis, we introduced a set of *get*()* functions that collect data from different types of knowledge. An example is the *getRequired()* function in the mobile learning case.

```
bool needAdaptationAdd(){
    return add_loc[RiD] == RECVD || enable_lane[RiD] == RECVD || add_dest[RiD] == RECVD;
}
```

Declaration 5. *Analyze.needAdaptationAdd* function for robot traffic application

The Analysis behaviour has three primary states. The result of the analysis is *Satisfied* when the managed system has the resources it requires to achieve its goals. The *Complete* state in the mobile learning case is an example. In the robots case, the satisfied condition corresponds to the transition *noNeedForAdaptation* that is taken when the robot uses the correct map. The analysis result is *Undersatisfied* when the system lacks resources to satisfy the current context and goals, and *Oversatisfied* when the system has redundant resources. These states directly match with *Redundant/Incomplete* and *ElementsExtra/ElementsMissing* from our illustrative scenarios. The results of the analysis are then communicated to the Plan behavior (e.g., *Plan_UnSatisfied[Node_ID]!*). Notice that an Analyze behavior can notify *Satisfied* states to the Plan behavior, which may be useful in case adaptation plans need to be cancelled. Fig. 10 shows the generic template for an analyze behavior with event triggering (*Analyze[Node_ID]?*).

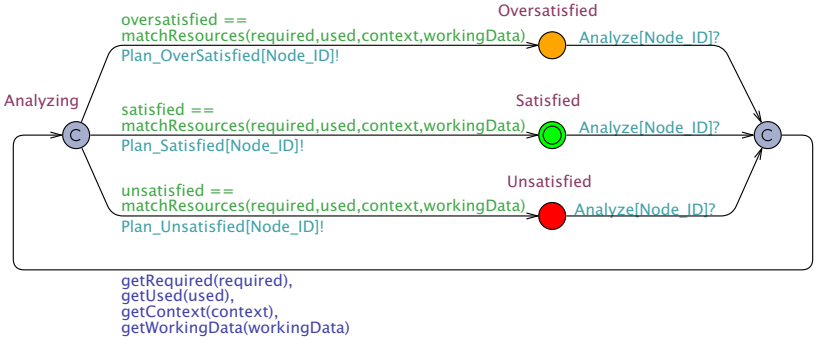

Fig. 10. Analyze behavior template with event-triggering

In summary, the Analyze behavior consists of the following steps: analyze triggering, data collection, analysis process and signalling the related plan behavior(s).

4.4. Plan

The Plan behavior is responsible for planning mitigation actions to adapt the managed system when needed.

4.4.1. Mobile Learning Case. Fig. 11 shows the specification for the Plan behavior. The primary responsibility of the Plan behavior is putting together plans to correct incomplete groups of mobile devices when requested (*SH_MVD_Incomplete[Mid]?*). The right hand side of the automaton defines the adaptation plan to recover the system from an undesired state. In particular, the *updateActions(Mid)* function defines actions to search for an available mobile device and integration the device in the group. The Plan behavior coordinates with the Execute behavior to apply the adaptation actions via the *addPhone[Mid]!* signal. On the left side of the automaton, the Plan behavior interrupts adaptation actions for particular cases in which the managed system has recovered from undesired states (i.e., when the activity requires fewer resources than before). Therefore, the Plan includes the option to cancel previous requested adaptation actions (*Execute_cancel[Mid]!*).

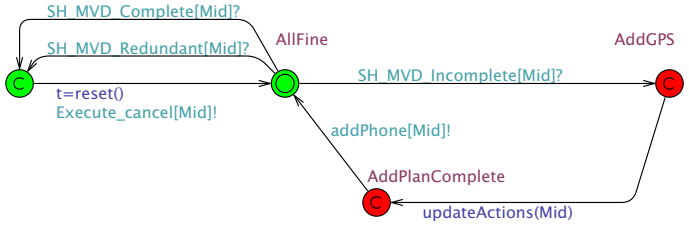

Fig. 11. Plan behavior for the mobile learning application

4.4.2. Robot Transportation Case. The plan behavior for the robotic case comprises two automata. The first automaton (Fig. 12-left) is responsible for composing a plan with adaptation actions to deactivate and remove items from the layout. The second (Fig. 12-right) composes a plan to activate and add items. The Plan behaviors are triggered by signals from the Analyze behavior. Then, the behavior identifies the type of required actions (e.g., *enableLane()* identifies the need to add a new lane in the layout) and plans the necessary adaptation tasks (*planEnabling()*). This process can be repeated multiple times until all the necessary adaptation actions are planned. Once planning is finished (*planned()* condition), Plan notifies the Execute behavior to execute the planned actions to the managed system (*execute*[RID]!*).

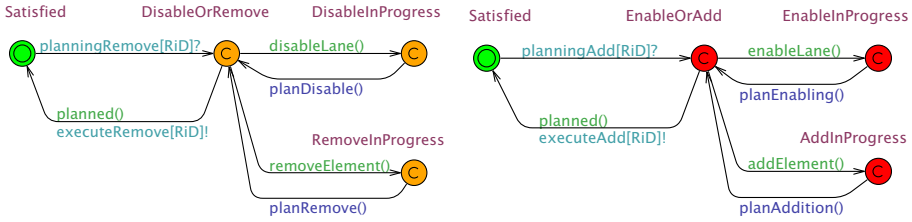

Fig. 12. Plan behaviors for the robot transportation application

4.4.3. Plan Behavior Template. Fig. 13 shows the Plan behavior template. The Plan behavior distinguishes between two types of mitigation actions depending on the results of an Analyze behavior (in case of a satisfied situation, no adaptation plan is required). Either there is a need to add resources (in case the system state is *unsatisfied*) or resources may be released (when the system state is *oversatisfied*). There are applications, such as the mobile learning case, in which oversatisfied scenarios may not require adaptations.

In the first case, (right part of the behavior template in Fig. 13), the Plan is triggered to compose a set of plan actions in order to add resources to the managed system (via the *addResource()* function). In particular cases, different planning behaviors may be specified to add resources, e.g., to

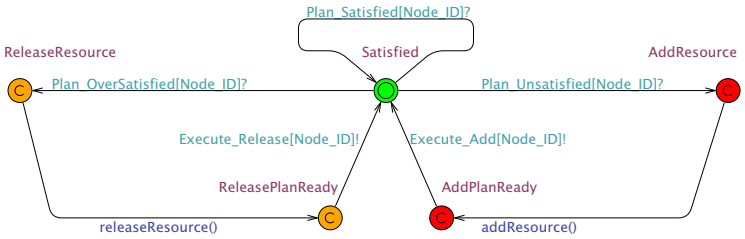

Fig. 13. Plan behavior template with time-triggering

model specific planning algorithms for different scenarios (such as enabling lanes or adding items to the robot layout). Afterwards, the Plan behavior notifies the Execute behavior to execute the adaptation actions on the managed system (*Execute_Add[Node_ID]!*). In the second case, resources can be released (left side of Fig. 13). The procedure to release a resource, which is specified in the *releaseResource()* function, is similar as for adding a resource.

In summary, we define the following steps of a Plan behavior: plan triggering, identification of the required type of plan, plan creation, and signaling execute behavior(s).

4.5. Execute

The Execute behavior is responsible for executing the adaptation actions of the generated plans.

4.5.1. Mobile Learning Case. The only required plan that needs to be executed in this case is handling *Incomplete* groups (see Fig. 14). Executing the plan requires looking for a mobile device and integrating it in the group (*Mid*). When triggered (*addPhone[Mid]?*), the behavior searches for an available device based on the criteria defined in the plan (*found_Phone=getFreePhone()*). If the search was unsuccessful (*found_Phone==NOPHONE*), the step is repeated every 2-time-units. Otherwise, the device that is found is integrated in the group (*Use[found_Phone]!*). The behavior is able to cancel the search process if required (*Execute_cancel[Mid]?*).

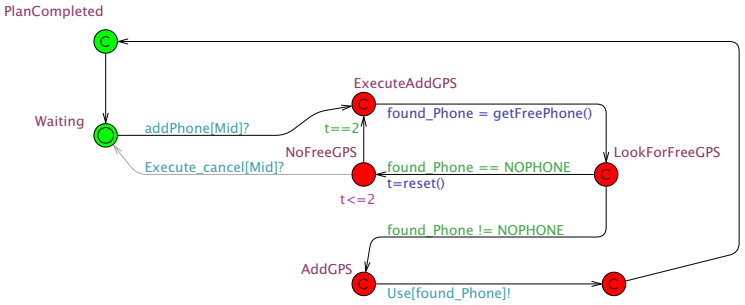

Fig. 14. Mobile learning Execute process

4.5.2. Robot Transportation Case. The Execute behavior for the robot transportation application is modeled with two automata. Fig. 15-top shows an automaton for enabling lanes and adding elements to the layout representation. For these scenarios, the behavior directly executes the adaptation plans. Possible options are enabling a lane, adding a location and adding a destination, which are represented in the Knowledge by *enable_lane*, *add_loc* and *add_dest* respectively. The state of these variables determines the planned action (*PLANNED*) that is selected for execution (e.g., *addLocation[Rid]!*), which triggers an Effector (shown in Fig. 16) that in turn specifies the function that needs to be applied to perform the action to the managed system. Finally, the Execute behavior marks in the Knowledge that the action is completed (e.g., *add_loc()*).

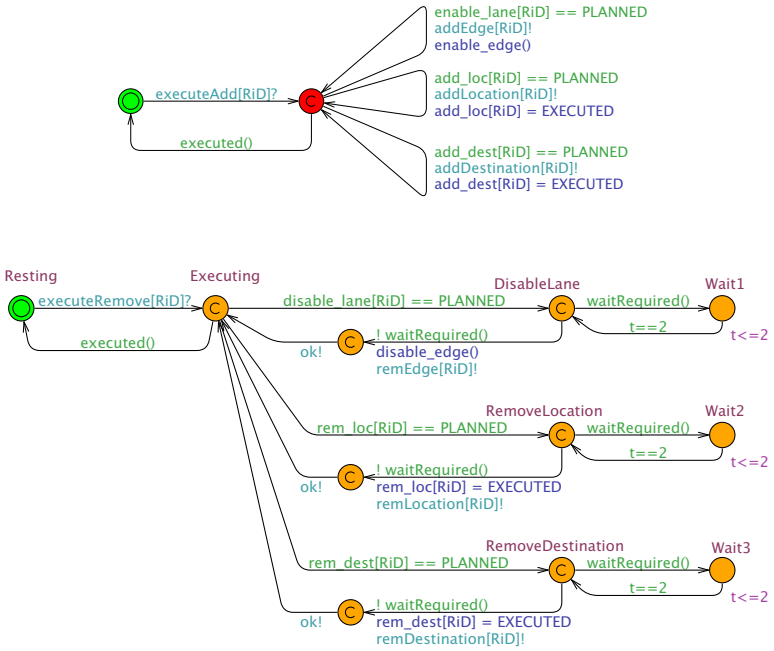

Fig. 15. Robot transportation Execute process

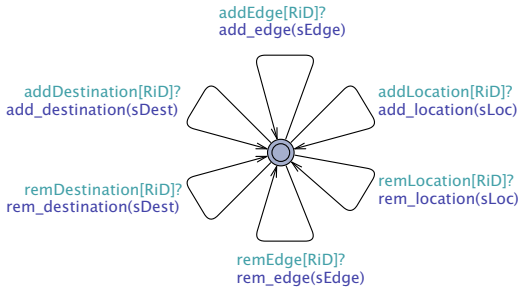

Fig. 16. Robot transportation Effector process

Fig. 15-bottom shows the execute behavior to remove and disable elements of the layout representation. The automaton identifies the type of actions to be executed, it coordinates with the managed system to prepare the adaptation actions, and it applies the adaptation actions on the managed system. The second step assures that no map elements are removed while the robot still requires them (e.g., *waitRequired()*, e.g. the robot should not disable a lane on which it drives).

4.5.3. Execute Behavior Template. Now we present the Execute behavior template that consolidates our expertise of designing Execute behaviors for different self-adaptive systems. The Execute behavior has two branches: one that deals with adding resources to the managed system; the other that deals with releasing resources. Fig. 17 shows the "add branch" of the Execute behavior template. The "remove branch" has a similar structure.

We identified three phases in an Execute behavior: preparation, execution (doAction) and post-execution. First, the Execute behavior is triggered to take action (*Execute_Add[Node_ID]?*). The behaviour checks whether the managed system is ready to invoke the planned actions (*isSystemReady(Prepare)*), an example is *waitRequired()* in the robot transportation case. If the system is not

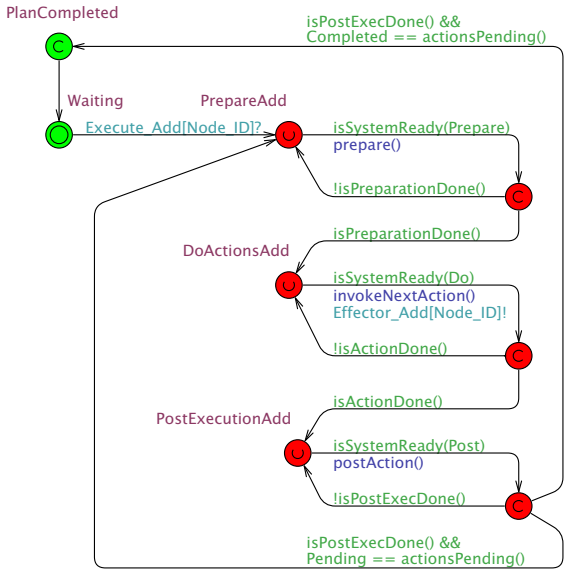

Fig. 17. Execute behavior template with time-triggering

ready, the behavior performs preparation tasks (*PrepareAdd* state) before executing the actions to add a resource to the managed system. Preparation typically includes steps such as locking certain resources (as in the mobile learning case), ensuring that the managed system is in a safe (quiescent) state, etc. Such preparation may require multiple preparation steps.

Once preparation is completed, the behavior moves to the *DoActionsAdd*, where adaptation actions are performed (such as integrating a new phone or activating a lane) via the (*invokeNextAction()* function and the *Effector_Add[Node_ID]! signal*.

Adding resources may require *PostExecutionAdd* tasks that are handled by the *postAction()* function. Examples are unlocking resources of the managed system and updating internal information in the Knowledge. Finally, the Execute behavior checks whether the plan is completed (*actionsPending()*). If more actions are *Pending*, the behavior returns to the *PrepareAdd* state from where it repeats the process to execute the next actions to the managed system.

5. PROPERTY SPECIFICATION TEMPLATES FOR VERIFYING MAPE-K FEEDBACK LOOPS

Timed computational tree logic (TCTL) allows designers to specify properties of the MAPE-K behaviors modeled in TA and verify whether the models comply with the properties. Properties specified in TCTL quantify over paths of the state space of the model and allow to verify reachability, safety, and liveness properties. During the design of the different self-adaptive systems, we defined a variety of properties. Before we present reusable property specification templates, we first illustrate some of the properties we specified for the two running cases.

In order to verify properties of the design of a self-adaptive system, it is important to note that the behavioral models of the MAPE-K feedback loop need to be integrated with models of the managed system and its environment. Section 3 provides some excerpts of the models for the mobile learning and the robot transportation cases; however, we do not further elaborate on the behaviors of the managed system and the environment as they are domain-specific and not the focus of this paper.

Mobile Learning Case

We discuss three properties in the mobile learning case that we used to verify the self-adaptive behavior.

```
G1: SAAnalyze(X).Incomplete --> SAAnalyze(X).Complete || SAAnalyze(X).Redundant
G2: SAAnalyze(X).Incomplete --> SAExecute(X).LookForFreeGPS
G3: SAAnalyze(X).Complete && ManagedSystemKnowledge[X].member[Y]==USED &&
      EnvironmentKnowledge[X].GPS[Y].state == DOWN --> SAAnalyze(X).Incomplete
```

Property G1 states that if an Analyze behavior detects that a group is incomplete and it needs a GPS service to satisfy the current requirements (state *Incomplete*), eventually the MAPE behaviors will solve the problem by adding a GPS service to the group (state *Complete* or *Redundant*) (the latter state may result for example when a GPS device is recovered during the adaptation process). This property verifies that the adaptation goals for the mobile learning case are achieved.

Property G2 allows states that whenever the Analyze behavior detects that a group is incomplete, the Execute behavior will eventually start looking for a GPS service (*LookForFreeGPS*). This property verifies the correct coordination and interactions between behaviors of the MAPE loop.

Finally, property G3 states that when an Analyzer finds a group that is complete, but one of the used resources (*ManagedSystemKnowledge[X].member[Y]==USED*, *Y* being the resource) is down (*EnvironmentKnowledge[X].GPS[Y].state = DOWN*), eventually the Analyzer will detect this failure and mark the Incomplete state (which will trigger the Plan behavior to look for a replacement of the GPS service). This property verifies the correct behavior of an internal MAPE behaviour.

Robot Traffic Case

We discuss two representative properties that we specified and verified for the robot traffic case.

```
R1: Analyze(X).ElementsMissing --> PlanEnableOrAdd(X).EnableOrAdd
R2: Monitor(X).DisableLaneRequest && disabledLane[X] == Lane_Z -->
      Execute(X).DisableLane && disabledLane[X] == Lane_Z
```

Property R1 states that when the Analyze detects that map elements are missing in the robot knowledge, the Plan will eventually add (*PlanEnableOrAdd* state in the Plan behavior) the missing parts of the map layout. This property focuses on guaranteeing correct interactions between Analyze and Plan behaviors, which is a critical aspect of the correctness of the self-adaptive behavior.

Property R2 states that when the Monitor behavior receives a request to disable a lane *Z*, that lane will eventually be disabled by the Execute behavior[3]. This property guarantees that the MAPE behaviors disable lanes correctly, which is one of the self-adaptation goals of this case.

5.1. Property Specification Templates

Now we present a set of generic properties for MAPE-K feedback loops that we derived from the design of the concrete self-adaptive systems we have developed. The property specification templates can be classified in three groups as shown in Fig. 18. Group 1, Adaptation Goal Specification Templates, includes properties related to the realisation of the adaptation goal by the MAPE-K feedback loop. Group 2, Intra-Behavior Specification Templates, specifies properties of internal MAPE component behaviors. Group 3, Inter-Behavior Specification Templates, focuses on properties that specify the interaction between multiple behaviors of a MAPE-K loop.

We provide a representative set of property specification templates of each group. For additional properties, we refer the interested reader to the project website.

5.1.1. Adaptation Goal Specification Templates. The adaptation goal specification templates focus on providing evidence about the realization of the self-adaptation goals. Therefore, they allow verifying that when a self-adaptive system requires adaptation, eventually the adaptation will be carried out. Examples of this type of templates are G1 and R2 in the illustrative scenarios.

```
P1: Analyze(X).Unsatisfied -->  Analyze(X).Satisfied || Analyze(X).Oversatisfied
```

P1 defines that when an Analyze behavior (Fig. 10) at node X detects that the managed system is in an unsatisfied state, the MAPE behaviors will eventually adapt the managed system bringing it

[3] *disabledLane[X]* is a global variable used to identify one (or more) specific lane(s) to be disabled

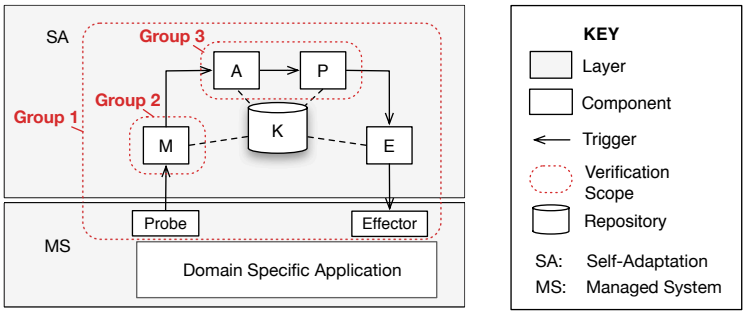

Fig. 18. Groups of property specification templates

in a satisfied or oversatisfied state. Property P1 is based on the assumption that eventually resources will become available to achieve the adaptation goal.

5.1.2. Intra-Behavior Specification Templates. Intra-behavior specification templates focus on verification of properties of individual MAPE behaviors, independently of the other MAPE behaviors.

```
P2: (ConcernKnowledge[X].required_resources >
      ManagedSystemKnowledge[X].used_resources) --> Analyze(X).Unsatisfied
P3: Plan(X).AddResource --> Plan(X).Satisfied
P4: Plan(X).ReleaseResource --> Plan(X).Satisfied
P5: Execute(X).DoActionAdd --> Execute(X).PlanCompleted
P6: Execute(X).DoActionRelease --> Execute(X).PlanCompleted
```

P2 generalises properties such as G3. P2 defines that when the resources used by the managed system are not sufficient according the requirements defined in the Knowledge, the Analyze behavior will eventually detect this Unsatisfied situation. The "greater than" symbol (>) is an abstract operator that checks whether the used resources satisfy the required resources (i.e., number of resources, quality of resources, etc.). This operator needs to be instantiated for the domain at hand. Intra-behavior properties are particularly relevant for complex behaviors. E.g., in complex adaptation scenarios, it is important to verify that the Plan behaviour creates adequate plans that will adapt the managed system as required. P3 and P4 are two property specification templates for a Plan behavior (Fig. 13) to verify whether adding (P3) and releasing a resource (P4) from the managed system leads to a Satisfied state. Similarly, P5 and P6 are specification templates for an Execute behavior (Fig. 17) to verify whether the plan for adding a resource to the managed system (P5) and releasing a resource from the managed system (P6) are correctly executed.

5.1.3. Inter-Behavior Specification Templates. Intra-behavior specification templates focus on interactions between MAPE behaviors; they are generalizations of properties such as R1 and G2.

```
P7: Analyze(X).Unsatisfied --> Plan(X).AddResource
P8: Plan(X).ReleasePlanReady --> Execute(X).DoActionRelease
P9: Plan(X).AddPlanReady --> Execute(X).DoActionAdd
```

P7 defines a property to verify the correct collaboration between the Analyze (Fig. 10) and Plan (Fig. 13) behaviors; i.e., when the Analyze behaviour detects an unsatisfactory state of the managed system, eventually the Plan behavior will create a plan to add a resource. P8 and P9 describe templates to verify the correct interaction between Plan and Execute (Fig. 17) behaviors. P8 specifies that when a Plan behaviour has generated a plan to release a resource, this plan is eventually executed by the Execute behavior. P9 specifies the correct execution of a plan to add a resource.

6. ASSESSMENT OF THE MAPE-K FORMAL TEMPLATES

To demonstrate the reusability of the MAPE-K Formal Templates for new applications in the target domain, we performed four case studies in the context of a course in a Software Engineering Master's degree at Linnaeus University. Through multiple case studies, it is possible to obtain qualitative evidence using techniques such as interviews, observations, and recordings [Adelman 1991; Miles and Huberman 1994; Yin 2009]. The aim of the case studies is to explore the usefulness and reusability of the templates, rather than aiming to be conclusive. The study results allow us to better understand and reason about how the participants used the templates in different cases. We focus on the setting of the case studies and the results. The complete material of the study is available at the project website [Gil de la Iglesia et al. 2014], including the protocol for the study, course materials, session recordings, and the detailed results.

6.1. Study Setting

The study was performed in the context of the *4DV108: Advanced Software Design* Master's course during a period of 6 weeks. The course included different phases on the design of a self-adaptive system with MAPE-K loops using formal methods.

We carried out four case studies. Each case study comprises a participant that designs and implements a self-adaptive system from a freely selected application domain which fits the characteristics of the *target domain* presented in Section 3. The selected application domains were *smart homes*, *security system* and *vehicular traffic systems*. Each case study is treated as a holistic unit with no independent embedded units [Yin 2009]. Thus, we take an holistic study approach with multiple-cases and study the data that is produced during the whole design and development process.

It is important to point out that the case studies are subject to a number of restrictions. In particular, the participants had no prior experience in the design of self-adaptive systems nor the use of formal methods. As such, we have to be careful with deriving conclusions about the effectiveness of the templates from the observations. However, as stated above, our primary goal of the case studies was explorative, aiming to get an insight in how the participants used the templates as well as the potential reuse of the templates.

Fig. 19 shows an overview of the course with the design of the study.

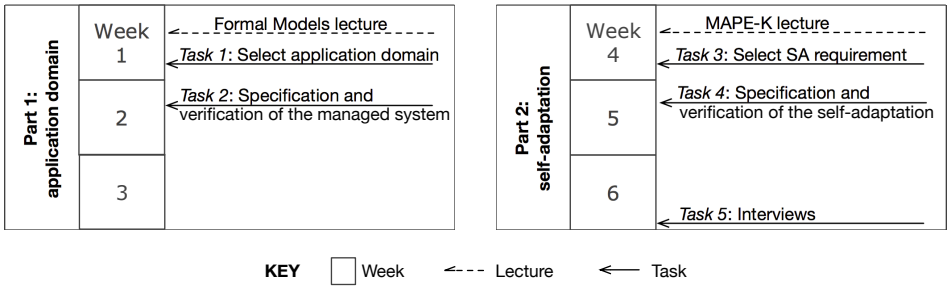

Fig. 19. Overview of 6-week course in which the study took place

The study included the following tasks that the participants had to perform:

Task 1. Select an application (managed system). The participants were not limited in the selection of the application domain (e.g., robotics, logistics, aeronautics, etc.) except that the application should comply with the characteristics of the *target domain* (described in Sec. 3)

Task 2. Design and specify the managed system behavior and required domain properties, and verify the properties.

Task 3. Select a self-adaptive requirement for the managed system that focuses on robustness or openness. The participants were not restricted in the specifics of the self-adaptation goal.
Task 4. Design and specify a self-adaptive solution for the managed system (design and required properties), and verify the correctness of properties of self-adaptive system.
Task 5. Participate in semi-structured interviews.

In Week 1, the participants were introduced to TA, TCTL and UPPAAL, the tool used for designing and verifying the formal models. In Week 4, the participants were introduced to the concept of self-adaptive systems, the MAPE-K reference model and the MAPE-K Formal Templates.

Throughout the study, the students were followed up closely by the course holders. After each step, we collected the designs to study the progression of the participants. At the end of the study, we carried out individual semi-structured interviews (Task 5) of about 30 minutes. The goal of the interview was to collect data about the participant experiences during the application design and the use of the templates in their domains.

6.1.1. Research Questions. The concrete aim of the study was to answer a set of research questions (RQ) that address reusability from different viewpoints.

RQ1. To what extent can the MAPE-K Templates be applied to applications that share the characteristics of the target domain?
RQ2. Do the Behavior Specification Templates allow the specification of MAPE-K behaviors for the specific self-adaptive systems, and if so, how?
RQ3. Do the Property Specification Templates allow the specification of properties for MAPE-K loops of the specific self-adaptive systems, and if so, how?

With RQ1, we wanted to provide initial evidence that the templates can be applied to additional applications in the *target domain*; i.e., applications that were not used as a basis from which the MAPE-K Formal Templates were derived. With RQ2, we wanted to analyze the reusability of the MAPE-K templates and to investigate the types of reuse of the behavior templates during the study. Similarly, with RQ3 we wanted to analyze the reuse of the MAPE-K property templates.

6.2. Analysis of the Results

We studied the design models of the different applications to identify the use of different parts of the behavior specification templates (states, transitions, functions, data structures) and property specification templates in the different application domains. Finally, we used the interviews to cross-check the results from the analysis process and complement them with first-hand experiences from the participants and feedback.

We present the results following the three research questions. We start with providing results about the reusability of the MAPE-K Formal Templates in different application domains. Then, we provide results w.r.t. reuse of the *Behavior specification templates* and we conclude with results for reuse of the *Property specification templates*.

6.2.1. Application Domains. During the study, the subjects designed self-adaptive systems from different application domains, including one *smart house system*, one *security system* and two *vehicular traffic systems*. Due to space limitations, we illustrate the analysis of the results with two of the four case studies. We selected two cases that provide a good and complementary coverage of the use of different aspects of the templates.

Case-1 is a smart house system that controls the heating system. The controller senses the temperature outside the house via a wireless channel, and it processes this information to adapt the heating regulator when needed. The controller was initially designed to work with one temperature sensor and one heating system regulator. When a second temperature sensor is added to the system, the controller would read data only from the last introduced sensor. To avoid biased data from a single temperature sensor, the home owner wanted the system to take into account temperature measures from multiple sensors around the house, which is an *openness* requirement. To that end,

Table I. Behavior Specification Template selection (left), and State selection per MAPE Behavior (right)

Template	Instances	Triggering
Probe	8	E
Monitor	6	E & T
Analyze	4	E & T
Plan	4	E
Execute	4	E
Effector	4	E

Template	Total States	Selected (%)	New
Monitor	30	15 (50%)	2
Analyze	20	15 (75%)	4
Plan	20	9 (45%)	0
Execute	56	11 (19.64%)	0
TOTAL	126	50 (39.68%)	6

a self-adaptation solution was applied to the managed system that enables the system to introduce new temperature sensors and incorporate their measurements.

Case-2 is a vehicular traffic application that controls traffic-lights to handle possible obstructions on the road; the obstructions can be sensed through traffic cameras. The quality of the cameras that identify obstacles depend on the scope of their view. Cameras have a higher precision with a narrow scope covering only one road. Therefore, two traffic cameras were placed on each crossing, in order to observe the traffic conditions from west and south sides providing a high degree of accuracy. The system was designed to notify drivers to take an alternative way ("turn right") using the traffic-lights when obstructions are detected. The goal of the self-adaptation in this case was to maintain a complete coverage of the roads when a camera failed by modifying the remaining camera settings in order to cover the west and south ways, at the cost of accuracy. This case specified a managing system that deal with system *robustness* to failing cameras.

Based on the collected data, we can answer research question RQ1 – *Can the MAPE-K Templates be applied to applications that share the characteristics of the target domain?* – positively as the templates were successfully applied to four different applications that fit in the target domain. We provide now a detailed analysis on the levels of reuse of MAPE-K Formal Templates in the study.

6.2.2. Reuse Behavior Specification Templates. In all the cases, the participants designed distinct automata for Probe, Monitor, Analyze, Plan, Execute and Effector behaviors. For all the cases, one automaton was specified for Analyze, Plan, Execute and Effector behaviors. For Monitor behaviors, more than one automaton on average were used. Self-adaptive systems usually require multiple data sources to monitor the managed system and its environment. We observed that multiple Probe behaviors were used to collect data (2 probes on average). The data is summarized in Table I-left.

We observed that event-triggered behaviors (*E* in Table I-left) were selected in all the cases, and complemented in some particular cases with time-triggering mechanisms (*T* in Table I-left) in order to specify time-out behaviors (e.g., Monitor and Analyze behaviors for Case-2). Fig. 20 shows an example where the behavior implements a time-out function using a time-triggering approach (*b==3* and *b<=3*). We will zoom in on some specifics of this behavior below.

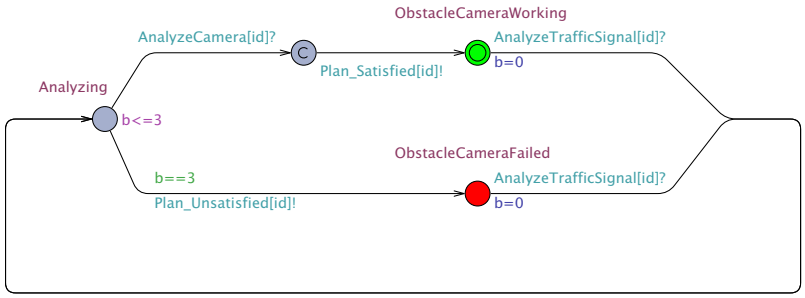

Fig. 20. Analyze behavior for a vehicular traffic system

The selection of event-triggering was explained as a particular requirement for the application domains. However, the complexity in modeling time-triggered behaviors was another factor for the

selection. One of the participants stated: *"Event-trigger is [the approach] I used the most. If I want to perform some actions, it is easy for me to send a signal. But time-triggering is hard to control."*

Regarding the reuse of the behavior specifications, one participant stated: *"I used all the templates, but I simplified them. I took away some states that were not necessary for the behavior, [but still] maintained the same logic"*. Reduction of the templates as a refinement approach was common to all the cases. Fig. 21 is an example of template refinement by reduction. It shows the *Monitor* behavior for the smart house system in which the *preProcessing* step has been removed. The *updateKnowledge* function has been refined (see Declaration 7) to update knowledge of the ManagedSystem represented by the *MSData* structure shown in Declaration 6. Concretely, the monitor is triggered when an additional temperature sensor is detected. The *updateKnowledge* function then registers the new sensor in the *ManagedSystemKnowledge*, which in turn allows the Analyze behavior to detect the need for self-adaptation.

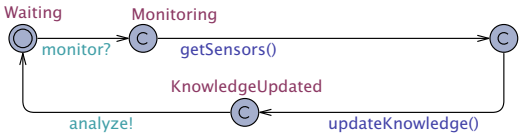

Fig. 21. Monitor behavior (Smart house system)

```
4  // Knowledge base SA
5  typedef struct  {
6      int  sensorID;
7      int  value;
8      int  failureflag  ;
9  }Data;
10
11 Data MSData [5]; //SA can support up to 5 sensors
```

Declaration 6. ManagedSystemKnowledge refinement (Smart house system)

```
4  void  updateKnowledge(){
5      for(i =0; i <5 ;i++){
6          MSData[i].sensorID = sens[i].sensorID;
7          MSData[i].value = sens[i]. value;
8      }
9  }
```

Declaration 7. Monitor functions refinement (Smart house system)

All the participants confirmed that they have based the design of their behaviors on the Behavior Specification Templates. We observed that the self-adaptation solutions that were required for the application domains did not require all the features provided by the Behavior Specification Templates. Therefore, the participants selected a subset of the states of the templates. This is particularly clear in the Plan and Execute behaviors, where only *add* branches were used, with no *preparation* and *postactions* states. Table I-right shows the level of reuse of states in the templates. For example, from the potential 30 states in Monitor specifications (5 states/Monitor x 6 Monitor behaviors specified), we count a total of 15 reused states, and 2 new states required for the behavior specification. For the specifications of Plan and Execute behaviors, only *add* branches were used.

We illustrate behavior specification template reuse and refinement with two other examples (we already gave one above for Monitor). Fig. 22-left shows the Plan behavior for the vehicular traffic application. In this case, the participant refined and reduced the Plan behavior template in order to cover the *addition* of a resource in the system (*UseCameraSouth*) when the west camera failed (*Plan_Unsatisfied[id]?*). In Fig. 22-right, we observe a refinement of the Execute behavior template for the smart house system. In this particular case, the Execute behavior calculates the average of the different temperature sensors via *calculateAV()* (which refines the *invokeNextAction()* function)

and feeds the controller of the heating system with the updated values (signal *updateValue!* to synchronize with the Effector behavior).

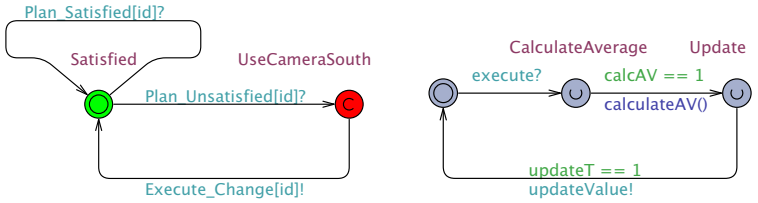

Fig. 22. Plan behavior for a vehicular traffic system system (left). Execute behavior for a smart house system (right)

In a small number of cases, the participants extended the behavior templates with additional states. We identified two reasons for these behavior extensions. First, intermediate states are used to refine parts of the existing behavior. Fig. 23 shows an instance of a state extension. In the Analyze behavior of the smart house system, the Unsatisfied state has been extended with two substates (*PD* and *PS*) in order to differentiate different types of undersatisfaction (when temperature sensors have different and equal values).

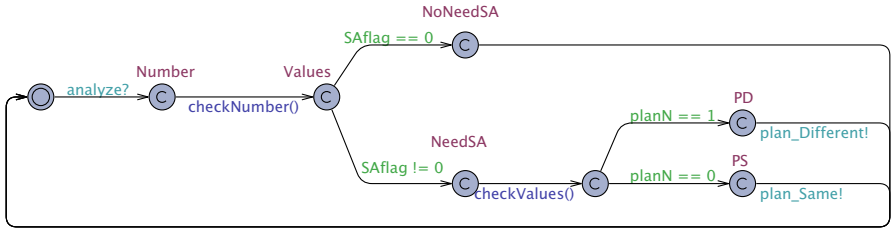

Fig. 23. Analyze behavior for a smart house system (right)

The second reason is the need for intermediate states in the behavior. The Analyze behavior in Fig. 20 illustrates this; the role of the *matchResources()* function is specified through a sequence of signals. When a camera is requested to provide the traffic conditions (*AnalyzeTrafficSignal[id]?*), it is expected to respond (*AnalyzeCamera[id]?*) in less than 3 time units (*b==3*). If accomplished, the system is considered to be in a Satisfied state (*ObstacleCameraWorking*). Otherwise, the camera is considered to be in a failed state (*ObstacleCameraFailed*), and the system is in an Undersatisfied state. To specify the sequence of signals, an additional state is introduced in the automaton.

This leads us to the analysis of reuse of functions in the behavior templates. We observed that only 20 from a total of 76 functions in the behavior templates were used for the specification of the MAPE behaviors (Table II). This selection aligns with the reduction of states (and transitions) used for specifying the behaviors. The selected functions were refined in order to specify the domain-specific part of the self-adaptation logic, usually via the specification of the functions.

However, we identified one particular case in which a function was replaced by a specified sequence of expected signals (time-out example in Fig. 20). We did not identify cases in which new functions were introduced to model MAPE-K behaviors.

The automata and the language primitives used to define functions and data structures can be considered as simplified C-like code. One of the interviewees indicated that the code used for the behavior specification may be highly reusable for the actual software implementation. However, another participant pointed to difficulties in transferring the design of signals and channels to software code. This confirms common sense that timed automata to formalize the MAPE-K behaviors have

Table II. Function selection per MAPE Behavior

Functions	Total	Selected (%)	New
Monitor	24	5 (20.83%)	0
Analyze	20	9 (45%)	0
Plan	4	1 (25%)	0
Execute	28	5 (17.86%)	0
TOTAL	76	20 (26.31%)	0

Table III. Adaptation Goal Property selection (left), Intra-behavior Property selection (center), and Inter-behavior Property selection (right)

Involved Behavior	Instances
Monitor	0
Analyze	3
Plan	2
Execute	0
TOTAL	3

Scope	Instances
Monitor	0
Analyze	5
Plan	1
Execute	0
TOTAL	6

Scope	Instances
Monitor-Analyze	1
Analyze-Plan	1
Plan-Execute	3
Execute-Effector	1
Plan-Effector	1
TOTAL	7

limitations in terms of the expressiveness, and some behavior abstractions (such as communication among distributed nodes) typically require more complex implementations.

Based on the results of our analysis, we can answer the second research question RQ2 – *Do the Behavior Specification Templates allow the behavior specification of MAPE-K components for the specific self-adaptive systems, and if so, how?* – positively. We observed that reuse of behavior templates was achieved via refinement, reduction, and the use of multiple instances of the templates. We observed different types of refinement, including refinement of states, transitions, functions (specification and logic) and knowledge structures.

6.2.3. Reuse Property Specification Templates. The participants used the property specification templates of the three groups to verify adaptation goals, intra-behavior relations and inter-behavior relations. In total, 16 properties were specified in the different case studies (see Table III).

Three properties were used to verify the overall adaptation goals. One of these properties was specified using the Analyze behavior only; the two others were specified using the Analyze and Plan behaviors. Six properties were used to verify the correctness of internal MAPE component behaviors. We observed that particular attention was given to verification of the correctness of the Analyze behavior (five cases; the sixth case related to the correctness of the Plan behavior). Finally, seven properties were used to verify the interaction between MAPE behaviors. Five of these properties involved the Plan behavior. In summary, we observe that the Analyze and Plan behaviors are considered as the most critical behaviors for self-adaptation by the participants. We illustrate each group of property specification templates with concrete instances.

The following properties are instances of Adaptation Goal Specification template P1:

```
P1: Analyze(X).Unsatisfied -->
      Analyze(X).Satisfied || Analyze(X).Oversatisfied

Pr1: Plan(1).UseCameraSouth --> Analyze(1).ObstacleCameraWorking
Pr2: AlarmAnalyze(0).NotWorked --> AlarmAnalyze(0).Worked
Pr3: Room.SmokeIncreased and Alarm(0).Broken -->
       Alarm(1).Red_loud  and Alarm(2).Red_loud
```

Pr1 specifies a property for a vehicular traffic system that verifies that when the West camera failed and the planner plans to use the South camera to detect obstacles (*Plan(1).UseCameraSouth*) the Analyzer will eventually reach a Satisfied state (as a result of the adaptation process), that is, the obstacle camera is working properly (*Analyze(1).ObstacleCameraWorking*). *Pr2* and *Pr3* belong to a third case study focusing on a security system for fire detection. Property *Pr2* specifies a property for a fire detection system that verifies that when the Alarm Analyzer detects that the detection

system is not working properly (*AlarmAnalyze(0).NotWorked*), this problem will eventually be resolved (*AlarmAnalyze(0).Worked*). Property *Pr3* specifies that when increasing smoke is detected and an alarm in one of room is broken, the alarms in the other rooms are directly put in highest alarm (*Alarm(x).Red_loud*).

The following properties are instances of Intra-Behavior Specification template P2:

```
P2: (ConcernKnowledge[X].required_resources >
      ManagedSystemKnowledge[X].used_resources) --> Analyze(X).Unsatisfied

Pr4: A[] (Analyzer.NoNeedSA imply numberOfSensors == 1)
Pr5: A[] (Analyzer.NeedSA   imply numberOfSensors  > 1)
```

Pr4 and *Pr5* specify properties for the smart house system that allow verifying that the Analyzer correctly detects the need for self-adaptation. Concretely, if only a single temperature sensor is registered (*numberOfSensors == 1*), there is no need for adaptation (*Analyzer.NoNeedSA*), but if more sensors are registered (*numberOfSensors* $>$ *1*) adaptation is required (*Analyzer.NeedSA*).

The following properties are instances of an Inter-Behavior Specification template P9:

```
P9: Plan(X).AddPlanReady --> Execute(X).DoActionAdd

Pr6: Planner.PlanReady --> Executor.CalculateAverage
Pr7: Execute(1).InformCameraSouth --> Effector(1).ChangeCameraAngle
```

Pr6 specifies specifies a property for the smart house system that verifies that when the planner is ready to incorporate a new sensor (*Planner.PlanReady*) the executer will eventually calculate the average of the sensed values of the sensors (*Executor.CalculateAverage*) and adapt the managed system accordingly (see Fig. 22-right). *Pr7* specifies a property that involves the Execute and Effector behavior for a vehicular traffic system. The property verifies that when the alternative south camera is activated to extend its monitoring scope (*Execute(1).InformCameraSouth*), the effector will expand the scope-view of the camera (*Effector(1).ChangeCameraAngle*) to cover west traffic (in addition to the south traffic covered in normal conditions).

The case studies demonstrate the reusability of the Property Specification Templates. During the interviews, the participants acknowledged that they directly used or based their property specifications on the Property Specification Templates. The templates assisted the participants to identify properties that were suitable and relevant to verify the correct behavior of their self-adaptive system.

Based on the results of our analysis, we can answer the third research question RQ3 – *Do the Property Specification Templates allow the property specification for MAPE-K loops for the specific self-adaptive systems, and if so, how?* – positively. The Property Specification Templates provide guidance to the designer to identify concrete properties for the self-adaptive system at hand. The participants stated that the proposed set of templates covered their needs and the property templates simplify the verification tasks. One of the participants stated: *"The properties [aka. property specification templates] saved time"*. The participants used property specification templates of the three groups to verify the correctness of their self-adaptive systems. The selection was not equally distributed; Inter- and Intra-behavior properties were more frequently used than Adaptation Goal Specification properties. This observation reflects the fact that designers not only verify self-adaptation goals, but put significant emphasis on the correctness of MAPE behaviors and their interactions.

6.2.4. Reflection on the Study Results. We run four case studies in order to collect qualitative evidence for the reusability of the MAPE-K Formal Templates. The successful application of the templates to different applications of the target domain demonstrates the reusability of the templates.

The participants commonly supported the benefits of the templates to reason about the managing system. One participant stated: *"It makes the process clear. I know what the components must do. It also helped to understand the process expected from managing systems."* Additionally, we obtained indications that the use of the templates has a positive effect on the required efforts and time to design the managing systems. The specification of distinct MAPE behaviors allows easier comprehension of the component roles. One participant stated: *"The way [the templates] are written, they*

almost work for every type of system. They need some small customization. The whole idea was clear and it saved a lot of time. [Also] it help understanding how the system would work".

The participants raised concerns in transferring formal specifications into code implementation. Aspects such as communication between components are highly abstracted in the Behavior Specification Templates, which allow powerful reusability, but are more difficult to transfer into code.

In general, the participants perceived the use of the MAPE-K Formal Templates for the design of a self-adaptive system as a satisfactory experience. One of the participants phrased it: *"[I would use the templates again] if I want to design a self-adaptive system, because it makes things easier just to follow the process"*. Note that the statements of the participants should be considered from the perspective of Masters students with no previous experience in design and development of self-adaptive systems. However, they show the potential for the reusability of the templates by such developers. The study results provide initial evidence that defining reusable templates for the field of self-adaptive systems is promising, which is pointed out as a key challenge in [Filieri et al. 2012]. To get additional evidence for the reuse of the MAPE-K Formal Templates, the qualitative results of this study need to be complemented with additional quantitative experiments.

During these case studies, the participants were free to select application domains. This was a critical aspect in order to avoid internal threat to validity of the results. The design of the managed system was performed before introducing self-adaptation concepts, to study the reuse of the templates in quasi-real settings, in which a self-adaptive layer is created upon a legacy managed system design. The students were also free use the MAPE-K Formal Templates for the specification of the self-adaptive systems' behaviors. The goal of the course was to rigorously specify and verify the behaviors of a self-adaptive system, therefore alternative approaches could have been applied. The recordings of the sessions, interviews, specifications and documentation generated during the study are available at the project website.

7. CONCLUSIONS

A recent survey [Weyns et al. 2012a] identified a lack of consolidation of the formal design of self-adaptive systems. The templates presented in this paper consolidate our research efforts on the use of formal methods for the design and verification of a family of self-adaptive systems. The templates provide reusable design knowledge, which can be considered as a domain specific language that allows rigorous modeling and verification of the behaviors of MAPE-based feedback loops for a target domain of distributed applications in which self-adaptation is used for managing resources for robustness and openness requirements via adding and removing resources from the system.

We believe that documenting design expertise (such as the templates presented in this paper) is an important effort in the evolution of an engineering field (in our case the field of self-adaptive systems). Several researchers share this belief. The first roadmap paper of the community states: "Creating a catalogue of feedback types and self-adaption techniques is an important and likely fruitful endeavor our community must undertake." [Cheng et al. 2009]. Another prominent example in the related domain of software architecture is: "Mature engineering disciplines are characterized by handbooks and other reference materials that provide engineers with access to the systematic knowledge of the field. Cataloging architectural patterns is a first step in this direction." [Shaw and Clements 2006]. When defining the behavior and property specification templates we had to balance between generality and usability. More specific templates are potentially more expressive and provide designers more fine-grained elements for modeling and verification. However, more specific templates introduce more complexity, which may hamper their usability. It may also limit the applicability of the templates in terms of the target domain. To find the right balance we derived the templates from different applications and validated the results with new cases. The templates have shown to be useful for the specification of behaviors of different application scenarios as demonstrated in the case studies. Whether they are useful for other domains, requires further study.

Following the essentials of the MAPE-K reference model, the behavior templates support the design of the MAPE functions as distinct behaviors. Benefits of designing the distinct adaptation functions are: support for fine grained verification, increase of adaptation understandability and reduc-

tion of system's complexity; the latter was demonstrated in a recent empirical experiment [Weyns et al. 2013a]. The case studies reported in this paper support these claims, we observed a reduction in efforts, both in terms of time to understand the MAPE behaviors and their design for concrete applications.

However, our experiences have also identified some limitations in terms of expressiveness of modeling. Timed automata provide an intuitive modeling language, but, the language primitives primarily support modeling of behavioral aspects of designs. Furthermore, aspects such as messaging and communication protocols, which are commonly used in distributed systems, can be difficult to model with the language primitives of timed-automata. Therefore, in some cases it may become challenging to model the behavior of a distributed self-adaptive system.

In our future efforts, we consider the application of the MAPE-K Formal Templates to more complex self-adaptive systems. This will allow us to further test the appropriateness of both the behavior and property specification templates and, moreover, their applicability to larger-scale applications. We plan to focus on the knowledge part of the MAPE-K loop and define reusable model templates for particular sub-domains. In this research line, we plan to incorporate support for modeling and verifying probabilistic properties, which are essential to model uncertainties in self-adaptive systems. Finally, we also plan to study interacting MAPE-K loops which are essential to deal with multiple concerns and self-adaptation in decentralized settings.

REFERENCES

L. Adelman. 1991. Experiments, quasi-experiments, and case studies: a review of empirical methods for evaluating decision support systems. *IEEE Transactions on Systems, Man and Cybernetics* 21, 2 (1991), 293–301.

G. Behrmann, A. David, P. Pettersson, W. Yi, and M. Hendriks. 2006. UPPAAL 4.0. *In Quantitative Evaluation of Systems* (2006).

J. Bengtsson and W. Yi. 2004. Timed Automata: Semantics, Algorithms and Tools. In *In Lecture Notes on Concurrency and Petri Nets (LNCS 3098)*, W. Reisig and G. Rozenberg (Eds.). Springer-Verlag.

D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti. 2012. Specification patterns from research to industry: a case study in service-based applications. In *International Conference on Software Engineering*. IEEE, 968–976.

B. Cheng and others. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science, Vol. 5525. Springer, 1–26.

R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese. 2014. Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511). *Dagstuhl Reports* 3, 12 (2014).

R. de Lemos, H. Giese, H. Müller, and others. 2013. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *LNCS vol. 7475*. Springer.

S. Dobson and others. 2006. A Survey of Autonomic Communications. *ACM Trans. Auton. Adapt. Syst.* 1, 2 (Dec. 2006), 223–259. http://doi.acm.org/10.1145/1186778.1186782

M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. 1998. Property specification patterns for finite-state verification. In *Wworkshop on Formal methods in Software Practice*. ACM, 9.

G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, and B. Petrus. 2009. Architecture-driven self-adaptation and self-management in robotics systems. In *Proc. of SEAMS*.

N. Esfahani, E. Kouroshfar, and S. Malek. 2011. Taming uncertainty in self-adaptive software. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Eegineering*. ACM, 234–244. http://doi.acm.org/10.1145/2025113.2025147

N. Esfahani and S. Malek. 2010. On the role of architectural styles in improving the adaptation support of middleware platforms. In *4th European Conference on Software architecture*. Springer, 8.

A. Filieri, C. Ghezzi, and G. Tamburrelli. 2012. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Form. Asp. Comput.* 24, 2 (2012), 163–186.

D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. 2004. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *IEEE Computer* 37 (2004), 46–54. Issue 10.

H. Giese and W. Schäfer. 2013. Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML. In *Assurances for Self-Adaptive Systems*. LNCS, Vol. 7740. Springer.

D. Gil de la Iglesia. 2014. *A Formal Approach for Designing Distributed Self-Adaptive Systems*. Linnaeus University Dissertations.

D. Gil de la Iglesia, M.U. Iftikhar, and D. Weyns. 2014. Reusable Templates to Support the Design of MAPE-K Loops for Self-Adaptive Systems. In *homepage.lnu.se/ staff/daweaa/MAPE-K-Templates.htm*.

D. Gil de la Iglesia and D. Weyns. 2013. Guaranteeing robustness in a mobile learning application using formally verified MAPE loops. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 83–92.

Didac Gil de la Iglesia and Danny Weyns. 2013. SA-MAS: Self-adaptation to Enhance Software Qualities in Multi-agent Systems. In *International Conference on Autonomous Agents and Multi-agent Systems*. AAMAS'13, Saint Paul, USA.

H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D.A. Menascé. 2010. Software adaptation patterns for service-oriented architectures. In *ACM Symposium on Applied Computing (SAC '10)*. ACM, 8.

H. Gomaa and M. Hussein. 2004. Software reconfiguration patterns for dynamic evolution of software architectures. In *4th Working IEEE/IFIP Conference on Software Architecture, WICSA'04*.

L. Grunske. 2008. Specification patterns for probabilistic quality properties. In *International Conference on Software Engineering, ICSE '08*. ACM, 10.

T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. 1994. Symbolic Model Checking for Real-Time Systems. *Information and Computation* 111, 2 (1994), 193 – 244.

M.U. Iftikhar and D. Weyns. 2012. A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System. In *Foundations of Coordination Languages and Self Adaptation, FOCLASA'12*.

M.U. Iftikhar and D. Weyns. 2014a. ActivFORMS: Active Formal Models for Self-Adaptation. In *Software Engineering for Adaptive and Self-Managing Systems*. SEAMS'14, ACM.

M.U. Iftikhar and D. Weyns. 2014b. Assuring system goals under uncertainty with active formal models of self-adaptation. In *Companion International Conference on Software Engineering, ICSE'14*.

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

R. Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-Time Systems* 2, 4 (1990), 255–299.

J. Kramer and J. Magee. 2007. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 259–268.

J. Magee and T. Maibaum. 2006. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *International Workshop on Adaptive and Self-Managing Systems*. ACM, 30–36.

M.B. Miles and A.M. Huberman. 1994. *Qualitative data analysis: An expanded sourcebook*. SAGE.

P. Oreizy, N. Medvidovic, and R. Taylor. 1998. Architecture-based runtime software evolution. In *20th International Conference on Software engineering, ICSE'98*. IEEE.

A.J. Ramirez and B.H.C. Cheng. 2010. Design patterns for developing dynamically adaptive systems. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 49–58.

M. Shaw and P. Clements. 2006. The Golden Age of Software Architecture. *IEEE Software* 23, 2 (2006), 31–39.

G. Tamura, N.M. Villegas, H. Muller, J.P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezze, W. Schafer, L. Tahvildari, and K Wong. 2012. Towards practical runtime verification and validation of self-adaptive software systems. *Software Engineering for Self-Adaptive Systems II, Springer* (2012).

E. Vassev and M. Hinchey. 2009. ASSL: A software engineering approach to autonomic computing. *Computer* 42, 6 (2009), 90–93.

D. Weyns, M.U. Iftikhar, D. Gil de la Iglesia, and T. Ahmad. 2012a. A Survey of Formal Methods in Self-adaptive Systems. In *Fifth International C* Conference on Computer Science and Software Engineering (C3S2E '12)*. ACM, 13. http://doi.acm.org/10.1145/2347583.2347592

D. Weyns, M.U. Iftikhar, and J. Söderlund. 2013a. Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment. In *Software Engineering of Adaptive and Self-Managing Systems, SEAMS'13*.

Danny Weyns, M. Usman Iftikhar, and Joakim Söderlund. 2013b. Do External Feedback Loops Improve the Design of Self-adaptive Systems? A Controlled Experiment. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '13)*. IEEE Press, Piscataway, NJ, USA, 3–12. http://dl.acm.org/citation.cfm?id=2487336.2487341

D. Weyns, S. Malek, and J. Andersson. 2012b. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 7, 1 (2012), 8.

D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goeschka. 2013. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*. Springer, 76–107.

D. Weyns, S. Shevtsov, and S. Pllana. 2014. Providing Assurances for Self-Adaptation in a Mobile Digital Storytelling Application Using ActivFORMS. In *International Conference on Self-Adaptive and Self-Organizing Systems, SASO'14*.

Robert K Yin. 2009. *Case study research: Design and methods*. Vol. 5. SAGE.

J. Zhang and B. Cheng. 2006. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering, ICSE'06*. ACM.

J. Zhang, H.J. Goldsby, and B.H.C. Cheng. 2009. Modular Verification of Dynamically Adaptive Systems. In *8th ACM International Conference on Aspect-oriented Software Development, AOSD'09*.