Modelling, specifying and verifying self-adaptive systems instantiating MAPE patterns

Marwa Hachicha*, Riadh Ben Halima and Ahmed Hadj Kacem

REDCAD Laboratory,
University of Sfax,
B.P. 1173, Sfax, Tunisia
Email: marwa.hachicha@redcad.org
Email: riadh.benhalima@enis.tn
Email: ahmed.hadjkacem@fsegs.rnu.tn
*Corresponding author

Abstract: Self-adaptive systems are able to modify their behaviour and/or structure to deal with their continuously changing environment and internal dynamics. Adaptive systems are generally more difficult to design, specify and verify owing to their high complexity. Ensuring the correctness of the system adaptation logic is very crucial. This correctness also depends on the time associated with events. In this paper, we propose a refinement approach that aims first at modelling step-by-step self-adaptive systems that instantiate MAPE patterns for decentralised control in self-adaptive systems. Second, these models are then automatically translated into Event-B specifications that can be proved using the Rodin theorem prover. This formal specification provides a way to verify several relevant properties for self-adaptive systems. We distinguish between three classes of properties: adaptation, system and temporal properties. We illustrate our approach by modelling and verifying the forest fire detection system that exhibits a self-adaptive behaviour.

Keywords: self-adaptive systems; MAPE control loop; MAPE patterns; structural; behavioural; modelling; Event-B method; formal verification.


Biographical notes: Marwa Hachicha is a PhD student in Computer Science. She has obtained a Master degree in Computer Science in 2013 from the Faculty of Economics Sciences and Management of Sfax. She is a member of the Laboratory on Research on Development and Control of Distributed Applications (RedCAD). In her Master degree project, she has proposed a probabilistic approach for energy saving in wireless sensor network using monitoring values prediction. Currently, her main research areas include software engineering in self-adaptive systems.

Riadh Ben Halima received his Master degree in Computer Science from the University of Sfax, Tunisia, in 2004 and the PhD degree in Computer Science from Paul Sabatier University, Toulouse, France, in 2009. He is Associate Professor in the National Engineering School of Sfax and also Associate Researcher at ReDCAD Laboratory, Sfax, Tunisia. His main fields of interest are distributed adaptive software, QoS provisioning, and dynamic software architecture management.

Ahmed Hadj Kacem received his Master degree in Computer Science and his PhD degree from Paul Sabatier University, Toulouse, France, respectively in 1991 and 1995. He joined the University of Sfax (Tunisia) as an Assistant Professor in 1997, then as a Professor in 2007. He was the Head of the Computer Science Department at the Faculty of Economics Sciences and Management of Sfax from 2008 to 2014. Since September 2014, he is the Dean of the Faculty of Economics Sciences and Management of Sfax. He participated in the initiation of many graduate courses. He is a member of ReDCAD Laboratory.
1 Introduction

Modern advanced systems are required to continuously perceive important structural and dynamic changes in their contexts, and autonomously react to such changes (De Lemos et al., 2013; Cheng et al., 2009). They aim at achieving particular quality goals and ensuring the required functionality. Therefore, there is a need to have self-adaptive systems that are able to adapt their behaviour in response to their perception. For example, critical systems such as spacecraft navigational systems, need to be self-adaptive in order to avoid anomalies and ensure the required functionality at runtime by managing their behaviour or structure without human intervention.

Kephart and Chess (2003) proposed the notion of an autonomic element, in the form of a Monitoring-Analysis-Planning and Execution (MAPE) loop that controls a managed computing element.

When systems are large, complex, and heterogeneous, a single MAPE loop may not be sufficient to manage all the adaptations in a system, therefore, multiple MAPE loops may be introduced. In such self-adaptive systems, M, A, P, E components from multiple loops must coordinate to perform adaptation. Consequently, MAPE control loops must become first-class entities throughout the lifecycle of self-adaptive systems (Weyns et al., 2013). However, only few studies (De La Iglesia and Weyns, 2015; Luckey and Engels, 2013) have focused on modelling and analysing structural and behavioural features of MAPE designs, in particular for systems with multiple interacting MAPE control loops. Self-adaptation introduces additional complexity in a modelling process. In fact, it is hard to understand and maintain models for self-adaptive systems. Furthermore, the development of such self-adaptive systems is extremely challenging and demands new formal approaches to ensure the correctness of the system adaptation logic at the design time. The correctness of the system adaptation logic depends on the correctness of the MAPE control loop process and the time associated with events.

However, the survey in Weyns et al. (2012) shows that formally founded design models, which verify relevant properties for self-adaptive systems, are highly demanded. Nevertheless, before the execution stage, verifying the relevant properties (reachability, safety, temporal constraints, etc.) for self-adaptive systems as early as possible (i.e. from the specification stage), in order to guarantee the functional correctness of the system and its adaptation logic, has multiple benefits. Even though many research approaches do not propose specifying and verifying real-time constraints for self-adaptive systems, because quantitative temporal aspects are not taken into account.

Moreover, the work in Weyns et al. (2012) shows that using model checking and theorem proving to verify relevant properties has gained limited attention. Nevertheless, model checking technique suffers from the problem of state explosion, especially for large systems.

Modelling, specifying and verifying complex self-adaptive systems are very complicated especially when involving time constraints. Therefore, it is a hard task for software architects who delay their development, especially without a specific reusable design and dedicated development tools.

In this paper, our goal is to handle the problem of modelling self-adaptive systems with multiple control loops and the formal verification of the system adaptation logic at the design time.

1.1 Contribution

To do so, we propose a refinement based approach (Badeau and Amelot, 2005) to model, specify and verify self-adaptive systems. Our approach is divided into three phases:

Firstly, we propose modelling self-adaptive systems that instantiate MAPE architectural patterns (Master/slave, Coordinated control, Hierarchical control, Regional planner and Information sharing) proposed in Weyns et al. (2013). Patterns are an established approach that document systematic knowledge in a particular area. It describes a generic solution for a recurring design problem. The patterns proposed in Weyns et al. (2013) are derived from common knowledge in the field of self-adaptation and experiences of the authors with building self-adaptive systems. They describe the different ways of organising and designing self-adaptive systems based on multiple MAPE control loops.

Moreover, we develop a graphical editor (as an Eclipse plug-in) to assist designers in their architectural choice.

Indeed, after designing a self-adaptive system, the expert has to ensure the correctness of the system adaptation logic by verifying several properties, such as reachability, safety, deadlock and so on.

In the second step, we propose an automatic transformation of the designed self-adaptive system into Event-B (Abrial, 2010) specifications.

In order to decrease complexity, the translation to Event-B is achieved through many refinement steps while using a set of transformation rules. In other words, refinement enables us to gradually build a model by making it more and more precise. In Event-B, structural features are specified with several contexts $C_i$ where behavioural features are specified with several machines $M_i$.

Thirdly, the resulting Event-B specifications from the second step will then be imported into the Rodin theorem prover that supports the generation of proof obligations. These specifications are enriched with several properties to check the correctness of the system adaptation logic. A property that has been proved as correct in the first level must still be correct in the subsequent level.

A software environment supporting the different features of this approach, has been implemented and integrated as a plug-in in the open source Eclipse framework. Consequently, this work represents an extension of our previous work (Hachicha et al., 2016) which considers only modelling structural features for self-adaptive systems.

1.2 Organisation of the paper

The remainder of this paper is organised as follows. In Section 2, we provide some background information on Event-B method and describe MAPE patterns for decentralised
control in self-adaptive system. Then, in Section 3, we introduce the forest fire detection system as our case study to illustrate our proposed approach. Section 4 gives a short overview of our approach. Then, in Section 5, we present the modelling phase of self-adaptive systems that instantiate MAPE patterns and the results of applying this modelling phase to our case study. In Section 6, we present the transformation phase of the modelled self-adaptive system into Event-B specifications. Section 7 shows how to verify several properties for self-adaptive systems. In Section 8, we present the Eclipse plug-in that implements our approach. In Section 9, we examine the related work dealing with modelling, specifying and verifying self-adaptive systems. The last section concludes and gives future work directions.

2 Basic concepts and notations

This section introduces the Event-B method and MAPE patterns for decentralised control.

2.1 Event-B method

The Event-B (Abrial, 2010) modelling language defines mathematical structures into contexts and formal model of the system into machines. The context is defined by abstract sets, constants, and axioms which describe the properties of constants. An Event-B machine describes a reactive system by a set of invariant properties and a finite list of events modifying the state variables.

An invariant is defined as a predicate that holds in all reachable states. An event is decomposed into a guard that specifies under which circumstances it might occur. Moreover, an event contains some generalised substitutions called actions that define the state transition associated with the event.

A machine M may see a context C, this means that all the carrier sets and constants defined in C can be used in M. A context Ĉ can extend a context C, which means that all the properties defined in C are added to Ĉ. Besides, a machine M may refine a machine M.

Event-B uses a top-down refinement-based approach. The refinement of a specification enables enrich it in a step-by-step fashion. It provides a way to strengthen the invariants and add details to a model. It is used to transform an abstract model into a more concrete version by modifying the state definition.

2.2 MAPE patterns for decentralised control

According to the study in Weyns et al. (2013), a self-adaptive system is situated in an environment and consists of two layers: managing sub-system and managed sub-system. The managed subsystem comprises the application logic. While the managing subsystem comprises the adaptation logic that deals with one or more concerns.

In the MAPE control loop (Kephart and Chess, 2003), a component Monitor (M) collects monitoring data from the underlying managed system and the environment through probes of the managed system. Then, a component Analyser (A) performs data analysis to check whether an adaptation is required. If so, it triggers a component Planner (P) that composes a workflow of adaptation actions necessary to achieve the system goals. These actions are then carried out by a component Executor (E) through effectors of the managed system.

Five patterns are proposed in Weyns et al. (2013): coordinated control, information sharing, regional planner, master/slave and hierarchical control.

Coordinated control and information sharing are based on a fully decentralised approach. Master/slave, regional planner, and hierarchical control, are instead based on a hierarchical distribution model, where higher level MAPE components control subordinate MAPE components.

In the coordinated control pattern, a MAPE loop is associated with each part of the managed system which is under its direct control. In this pattern, all the M, A, P and E components of each loop coordinate their operation with the corresponding peers of the other loops. In contrast to the coordinated control pattern, the information sharing pattern restricts the inter-component interactions to M components only.

The master/slave design pattern models the centralised management approach in self-adaptive systems where control is centralised in a master component.

Regional planner pattern provides one P component (a regional planner) to collect the necessary information to plan adaptations. Regional planners interact with one another to coordinate adaptations that span multiple regions. Finally, the hierarchical control pattern structures the adaptation logic as a hierarchy of MAPE loops.

3 Our case study: the forest fire detection system (FFDS)

In the last couple of years, large wildfires such as the Fort McMurray wildfire in Alberta Canada (May 2016) have caused extended damages and catastrophic consequences, in loss of properties and lives. It has been deemed extreme by the Alberta Agriculture and Forestry website. This wildfire in the Fort McMurray area of Alberta has burned approximately 7,686 hectares (almost 19,000 acres) in a heavily populated area.

About 80% of homes were lost in Beacon Hill, serious loss in Abasand, waterways, etc. The magnitude of the disaster, in terms of death, property damage and cost, lead to think seriously about preventive solutions using wireless sensor networks. Wireless sensor networks (WSN) have been widely investigated in fire detection systems to detect and predict fire occurrence promptly and accurately. Each node in the network is equipped with a wide array of chemical sensors, on-board cameras, temperature sensors, humidity sensors, and other sensors. They periodically sense the environment to decide if there is an emergency situation or not (increase of temperature, a sudden deep decrease of humidity, the presence of smoke, and so on). When a significant change in temperature or smoke is detected by some sensor nodes, these nodes report packets which contain parameter measurements to a fire control centre for possible actions, such as alerting local residents and
dispatching firefighting crews, ambulances, aircraft, calling human emergency workers, etc.

Our FFDS is based on a WSN made by two different kinds of nodes: Node 1 (temperature sensor) and Node 2 (smoke detector) connected via the Internet to a fire control centre.

In an efficient detection system, it is not sufficient for the sensor node to notice the event: it should also have the energy to notify such an event quickly and accurately. In the forest fire detection system, introducing delay in reporting a fire can be regarded as a failure of the monitoring system and can have dramatic consequences. By monitoring the energy consumption, we are able to estimate when a sensor node will fail and then take the required actions when the battery level reaches a specific threshold.

In order to save energy, the following adaptation actions can be taken:

- Reduction of the monitoring frequency when the battery level is partially charged
- Turn off a node when the battery level is low
- Elect a new master when its battery level is very low

If the WSN fails to properly adapt its energy within a specific time, this can affect the functional correctness of the whole system.

In this paper, we propose the instantiation of two MAPE patterns: Master/Slave and coordinated control patterns. In the Master/Slave pattern, the fire control centre is elected as a master, while the two nodes serve as slaves. In the coordinated control pattern, each node is associated with a manager element containing a MAPE control loop that handles the adaptation of the managed element (i.e. node).

**Figure 1** The forest fire detection system

**Figure 2** Proposed approach
4 Our approach in a nutshell

In this section, we outline our modelling, transformation and verification process. Our approach is divided into three phases as shown in Figure 2.

Step-by-step modelling phase: As described in Figure 2, the designer chooses one of the provided self-adaptive design patterns in order to model his system. We offer a graphical editor as an Eclipse plug-in to assist designers in their architectural choice. Five patterns are available in our modelling tool.

Automatic generation of Event-B specifications phase: At each refinement step, our tool generates an XML file and transforms it according to transformation rules expressed with the XSLT language (eXtensible Stylesheet Language Transformations) into Event-B specifications. These transformation rules are applied to a source XML document (self-adaptive system description model) to obtain a new result document containing Event-B specifications. We have to mention that our contributions are not limited to these two phases, but we choose to undertake the verification of the correctness of the system adaptation logic using the Event-B method.

Verification process phase: The resulting Event-B specifications will then be imported into the Rodin (Abrial et al., 2010) theorem prover that supports the generation of Proof Obligations belonging to Event-B models. A proof obligation is something that has to be proven to show the consistency of the machine, the correctness of theorems, etc. Thereby, in this paper, we distinguish between three classes of properties: system, adaptation and temporal properties. These properties are described using invariants or guards in events in the machine part of Event-B method. All proof obligations should be discharged to prevent the different properties from being violated.

In the next sections, we provide a description of our proposed approach.

5 Step-by-step self-adaptive system modelling

We provide a modelling solution to describe the self-adaptive systems that instantiate the MAPE patterns proposed by Weyns et al. (2013) using a visual notation based on our UML profile.

5.1 Structural features

In the structural modelling step, we propose a UML profile that extends the UML 2.0 component meta-model. The main participants that make up the architecture of a self-adaptive system that instantiates a MAPE pattern can be master, slave, managed element, manager element or regional planner.

In this paper, we particularly present the instantiation of two MAPE patterns: master/slave and coordinated control patterns. The conceptual model of the master/slave pattern is shown in Figure 3. The conceptual model of the coordinated control pattern is shown in Figure 4.
The entities that make up the architecture of a master/slave pattern are <<Master>> and <<Slave>>. They represent a sub-class of UML package meta-model. There is a single instance of the master element containing a group with a P and an A component, and there can be an arbitrary number of instances of slave elements of the group with an M and an E component. The <<MAPEComponent>> represents a sub-class of UML component meta-model. The <<Monitor>>, the <<Analyser>>, the <<Planner>> and the <<Executor>> extend the <<MAPEComponent>>. The slave has one or several <<Probe>> and <<Effector>>. The <<Probe>> makes measurements about a <<ContextElement>> with the purpose of sensing a specific variable of interest during runtime. The <<Effector>> is configured to effect changes needed to alter the target of the system behaviour according to adaptation needs. We noticed that each MAPE component contains three sub-components, namely a receiver, a processor and a sender.

- The <<Sender>> sub-component publishes symptoms, request for change (RFC), plans, etc.
- The <<Processor>> sub-component aggregates and filters events, analyses data, processes plan, etc.
- The <<Receiver>> sub-component receives information and monitoring data.

The <<Receiver>> and <<Processor>> are connected through the \( R = P \) dependency. The <<Processor>> and <<Sender>> are connected through the \( P = S \) dependency.

The main entities that constitute the architecture of a coordinated control pattern are the <<Manager>> and the <<ManagedElement>>. They represent a sub-class of UML package meta-model. A MAPE loop is associated with each managed Element. The managed element comprises the application logic that provides the system domain functionality. It contains effector and probe components to perceive and affect the environment. The manager contains all the MAPE components. Each MAPE component can interact with its peers to share particular information or coordinate adaptation actions. For example, the Analyser components exchange information to make decisions about the need for an adaptation. Consequently, we add an association <<MonitoringInfInteraction>> that represents the interactions between the Monitor components of two different managers. The <<AnalysingInfInteraction>> association represents the interactions between the Analyser components of two different managers. The interactions between the Planner components are presented using the <<PlanningInfInteraction>> association.

The <<ExecInfInteraction>> association represents the interactions between the Executor components of two different managers.

We start constructing an initial model by describing only the main purpose of our system. Then, other details such as adding new MAPE sub-components, new connections between two components, etc. can be gradually introduced into subsequent concrete models.

In the first level of the Master/Slave pattern instantiation, we create a very abstract model composed of a master (connected to a fire control centre) and two slave components (Node1, Node2). The slave contains the Monitor and the Executor components. The master contains the Analyser and the Planner components. In this step, we concentrate on the communication between the different entities (master and slaves).

In the second level, we add the different MAPE sub-components (sender, processor and receiver) and their connections.
In the third level, as shown in Figure 5, we introduce the \textit{ContextElement}, the \textit{Probe}, the \textit{Effectors} components and their different connections.

In the first level of the coordinated control pattern instantiation, we create a very abstract model composed of two abstract managed elements (nodes) and two managers. Each manager contains a MAPE loop. In this step, we concentrate on the communication between the different MAPE components.

In the second level, we add the different MAPE sub-components and their connections.

Finally, as shown in Figure 6, we model the different components of each managed element and the different connections with their manager elements.

5.2 Behavioural features

We provide a modelling solution to describe step-by-step behavioural features of self-adaptive systems using a UML activity-based profile.

The master/slave pattern (Weyns et al., 2013) is characterised by the presence of a global MAPE control loop, which manages a higher-level adaptation of behaviour of multiple slave components. In the first modelling level of the FFDS instantiating the master/slave pattern, we create a very abstract model composed of two activities: monitoring and executing in the slave element and two activities: analysis and planning in the master element. Then, in the second level, we add the different MAPE sub-activities like collect battery level symptom, interpret battery level, etc.

Finally, as shown in Figure 7, we add activities related to probe and effector components. The probe components provide an activity to measure the battery charge. These measurements are then collected and filtered by the monitor phase. Then, the analysis phase (in the fire control centre) of the MAPE control loop gathers and interprets them. The analyser sends a request for change to the planner if there is degradation in the battery level. If the awareness level is not satisfied, then the planning phase triggers and devises a plan to execute awareness change. To achieve this, the execute phase of the two slaves notifies effectors which adapt the required awareness level (reduce monitoring frequency or switch off a node).

Figure 5 Master/Slave pattern applied to the FFDS use case: structural modelling (level3)
Figure 6  Coordinated control pattern applied to the FFDS use case: structural modelling (level 3)

Figure 7  Master/Slave pattern applied to the FFDS use case: Behavioural modelling (level 3)
The coordinated control pattern (Weyns et al., 2013) is characterised by the presence of a MAPE control loop associated with each managed element. Besides, the coordinated control pattern is characterised by the coordination of all the M, A, P, E components of each loop in the manager element with corresponding peers of other loops.

In the first modeling level of the FFDS instantiating the coordinated control pattern, we create a very abstract model composed of four elements: a manager element Node1Manager associated with Node1 and a manager element Node2Manager associated with Node2. The different M, A, P, E components of the manager Node1Manager coordinate their operations with corresponding peers of manager Node2Manager. The M component of manager Node1Manager interacts with the M component of manager Node2Manager to collect energy measurement. Thus, the information collected about the status of Node1 is shared with the manager of Node2. The A component of manager Node1Manager exchanges information with the A component of manager Node2Manager to make a decision about the need for an adaptation. Moreover, the P component of manager Node1Manager interacts with the P component of manager Node2Manager by sending the analysis report to avoid antagonistic adaptation. The E component of manager Node1Manager exchanges messages with E component of manager Node2Manager to synchronise adaptation actions.

Then, in the second level, we add the different MAPE sub-activities.

Finally, as shown in Figure 8, we add activities related to probe and effector components in the different nodes.

Figure 8 Coordinated control pattern applied to the FFDS use case: Behavioural modelling (level 3)
6 Automatic generation of Event-B specifications

The second step of our approach consists in translating the graphical model of a self-adaptive system into the formal notation of Event-B by applying a set of transformation rules. Our choice of Event-B method is motivated by the fact that Event-B offers a theorem prover. Additionally, it enables to represent systems at different abstraction levels. Therefore, it provides a way to manage design, prove complexity, and reduce efforts during the development. Besides, the concept of refinement is the main feature of Event-B. The refinement of a machine allows to enrich it in a step-by-step fashion. It is the foundation of the correct-by-construction approach. An Event-B specification is considered correct only if each machine, as well as the process of refinement, are proved by adequate theorems named Proof Obligations.

In what follows, we present the major rules allowing the transformation of a graphical model into an Event-B specification.

6.1 Structural features rules

Pattern entity transformation rule: The entities that constitute the architecture of a self-adaptive system can be a master, a slave, a managed element, a manager element, monitor components, etc. This rule transforms each entity name (master, slave, managed element, manager element, regional planner, monitor, analyser, etc.) into constants in the CONSTANTS clause.

The set of entities is composed of all entity names. This is transformed formally into a partition: \((\text{EntityName} - \text{partition})\):

\[
\text{Entity} = \{\text{Entity}_1, ..., \text{Entity}_n\} \land \text{Entity}_1 \neq \text{Entity}_2 \land ... \land \text{Entity}_{n-1} \neq \text{Entity}_n.
\]

The specification of the different main entities (master and slaves) in our case study is presented as follows:

<table>
<thead>
<tr>
<th>CONSTANTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptationActions</td>
</tr>
<tr>
<td>AnalysisInformation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AXIOMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptationActions = Relation : AdaptationActions ∈ Planner − Components \leftrightarrow Executor − Components</td>
</tr>
<tr>
<td>partition(AdaptationActions, ReduceMF, TurnOFF)</td>
</tr>
<tr>
<td>ReduceMF = Domain : dom(ReduceMF) = PS</td>
</tr>
<tr>
<td>ReduceMF = Ran : ran(ReduceMF) = PR</td>
</tr>
<tr>
<td>MonitoringInformation = Relation : MonitoringInformation ∈ Monitoring − Components \leftrightarrow Monitoring − Components partition(MonitoringInformation, EnergyMeasurement)</td>
</tr>
<tr>
<td>EnergyMeasurement = Domain : dom(EnergyMeasurement) = SS</td>
</tr>
<tr>
<td>EnergyMeasurement = Ran : ran(EnergyMeasurement) = EC</td>
</tr>
</tbody>
</table>

6.2 Behavioural features rules

The activity diagram is a way to describe the dynamic behaviour of a system. It is specified with the Event-B method in the MACHINE part. A machine of a system specification \(M_i\) contains a number of events and has a state defined by means of a number of variables and invariants. From this diagram, we can extract for each Activity Partition (it contains a group of actions) the Activity_ Partition_Name, and Action_Name.

Event generation rule: This rule transforms each action in a partition (monitoring, analysis, planning, execute, send, process, receive) into an event. We specify the first abstract FFDS model with a machine at a high level of abstraction. Then, we add all the necessary details to the first machine by using the refinement process.

Abstract Machine. For instance, we transform the action named Monitoring into an Event: Event Monitoring. In the abstract machine, this event just updates the following task of the slave element (Act1).
Refined Machine. For each adaptation action in the generated plan, an event name \texttt{Execute-Action-i-name} is generated. Each event refines the event \texttt{Execute} of the abstract machine. In our case, we have two adaptation actions for our plan. Therefore, two events are generated: \texttt{Execute TurnOFF} and \texttt{Execute ReduceMF}. The specification of the event \texttt{Execute ReduceMF} is shown below.

\begin{verbatim}
EVENT ExecuteReduceMF
REFINES EVENT Execute
ANY slave MF
WHERE
Grd1: \text{slave} \in \text{Slaves}
Grd2: \text{Stepslave(slave)} = \text{Execute}
Grd3: MF \in \text{M}
Grd4: \text{SlaveAction(slave)} = \text{ReduceMF}
Grd5: \text{EffectorState(slave) \rightarrow ReduceMF} = \text{Active}
THEN
Act1: \text{Stepslave(slave)} := \text{Execute}
Act2: \text{SlaveMonitoringFrequency(slave)} := MF
END
\end{verbatim}

Variables generation rule: This rule extracts from \texttt{Activity-PartitionName} the set of the variables. Some of variables can be general like the variable \texttt{Stepmaster}, which denotes the different steps of the master element and the variable \texttt{RequestForChange} which denotes the request for change observed by the master or the manager element.

\textbf{Invariant generation rule:} Invariants that define the variables can be automatically generated. For example, the variable \texttt{Stepslave} is defined with the following invariant: \texttt{Stepslave \in Slaves \rightarrow Stepslave} which specify that \texttt{Stepslave} is a relation between the slave element and its different steps (Monitoring, Execute).

7 Verification process

Based on the formal specification described in section 6, we are able to verify a set of properties using the Event-B method to check the system adaptation logic is correct. In this section, we propose the verification of three classes of properties: adaptation properties, system properties and temporal properties.

7.1 Adaptation properties

These general properties are related to adaptation where any self-adaptive system should satisfy them. They do not depend on the specific modelled system.

7.1.1 Safety properties

Safety properties assert that nothing bad should happen during the program execution. For example, ensuring mutual exclusion, deadlock freedom, preserving invariants after a system reconfiguration, access control of resources, etc.

\textbf{Deadlock freedom:}

- The master or the manager element must not be blocked in the plan step, i.e. planned actions must be executed by the slave components or managed element. This is controlled by the following invariant property in the Master/slave pattern:
  
  \textbf{Invariant 1:} \phantom{a} \forall M \cdot M \in \text{Master} \land \text{Action} \land i \in \text{Actions} \land \text{Planning}(M, \text{Action} \land i) \Rightarrow \exists S \cdot S \in \text{Slaves} \land \text{Execute}(S, \text{Action} \land i)

Where \texttt{Planning}(M, \text{Action} \land i) and \texttt{Execute}(S, \text{Action} \land i) are Planning and Execute events in the machine part respectively.

\textbf{Error process:}

- In the Master/slave pattern, the effector of any slave o must not be active if the request for change observed by the master is False.
  
  \textbf{Invariant 2:} \phantom{a} \forall M, S \cdot (M \in \text{Master} \land S \in \text{Slaves}) \Rightarrow \neg (\text{RequestForChange}(M) = \text{FALSE} \land \text{EffectorState}(S) = \text{Active})

- In the coordinated control pattern, the effector of any managed element must not be active if the request for change observed by the master element is False.
  
  \textbf{Invariant 3:} \phantom{a} \forall ME, AM \cdot (ME \in \text{ManagedElement} \land \text{AssocManager}(ME) = AM) \Rightarrow \neg (\text{RequestForChange}(AM) = \text{FALSE} \land \text{EffectorState}(ME) = \text{Active})

- The slave must execute actions planned by the master. An error occurs if the slave executes another action.
  
  \textbf{Invariant 4:} \phantom{a} \exists M, S, a \cdot (M \in \text{Master} \land S \in \text{Slavea} \land a \in \text{Actions} \land \text{SlaveAction}(S) = a) \Rightarrow (\text{EffectorState}(S \rightarrow a) = \text{Active})

\textbf{Mutual Exclusion:}

- In the coordinated control pattern, two different executor components must never send commands to the same effector of a managed element at the same time.
  
  \textbf{Invariant 5:} \phantom{a} \forall E1, E2, ef \cdot t1, t2 \cdot (E1 \in \text{Executor Components} \land \text{ef} \in \text{Efectors} \land t1 \in \text{Time} \land t2 \in \text{Time} \land t1 = t2) \Rightarrow (E1 \rightarrow ef) \notin \text{Commands} \land (E2 \rightarrow ef) \notin \text{Commands})

7.1.2 Reachability properties

A reachability property asserts that a system should reach a particular state.
7.2 System properties

These properties go into finer detail and require specific knowledge on the system to be analysed.

- In the Master/slave pattern, the probe of each slave should reach the enabled state when the slave operates in the monitoring step.
  
  Invariant 6: \( \forall S \cdot S \in \text{Slaves} \land \text{StepSlave}(s) = \text{Monitoring} \Rightarrow \text{ProbeState}(s) = \text{Enabled} \)

- In the coordinated control pattern, the probe of each managed element should reach the enabled state when the associated manager element operates in the monitoring step.
  
  Invariant 7: \( \forall ME, AM \cdot ME \in \text{ManagedElement} \land AM \in \text{ManagerElement} \land \text{AssocManager}(ME) = AM \land \text{StepManager}(AM) = \text{Execute} \Rightarrow \text{EffectorState}(s) \rightarrow \text{SlaveAction}(s) = \text{Active} \)

- In the coordinated control pattern, the effector of each managed element should reach the activated state when the slave operates in the execution step.
  
  Invariant 8: \( \forall S \cdot S \in \text{Slaves} \land \text{StepSlave}(s) = \text{Execute} \Rightarrow \text{EffectorState}(s) \rightarrow \text{SlaveAction}(s) = \text{Active} \)

- In the coordinated control pattern, the effector of each managed element should reach the activated state when the associated manager element operates in the execution step.
  
  Invariant 9: \( \forall ME, AM \cdot ME \in \text{ManagedElement} \land AM \in \text{ManagerElement} \land \text{AssocManager}(ME) = AM \land \text{StepManager}(AM) = \text{Execute} \Rightarrow \text{EffectorState}(s) \rightarrow \text{SlaveAction}(s) = \text{Active} \)

7.3 Temporal properties

Temporal properties are useful to check whether the different events are carried out within specific deadlines and met temporal constraints (Cansell et al., 2007). We identify four main categories of temporal properties for self-adaptive system: delay, temporal constraint over cardinality, stabilisation and expiry.

Delay Time: A given occurred event of the MAPE control loop must start after the execution of the previous event to avoid overlapping between different events.

Let \( T_M \) be the starting time of the monitoring event. Let \( T_a \) be the starting time of the analysis event and let \( D \) be the duration of the event monitoring. The Delay time constraint is defined as: \( T_a > T_M + D \). There are two steps in order to add a delay constraint in our Event-B specification. First, the occurrence time of the monitoring event is recorded in a variable \( T_m \). Then, in the event analysis, which should be delayed, a guard (Grd2) is needed which forces the event to be eligible to occur after the stated delay period \( D \) has been passed from the occurrence of the trigger event. Moreover, we add the event Tick-Tock which progresses time (Act1) in an Event-B model.

A general pattern of the two events (monitoring and analysis) with the delay property is shown below.

```
EVENT Monitoring  EVENT Tick – Tock
WHERE
Grd1 : G(M)  Act1 : time := time + 1
THEN
Act2 : Act(M)
END

EVENT Analysis
WHERE
Grd1 : G(A)
Grd2 : T_a > T_m + D
THEN
Act1 : Act(A)
END
```

Temporal constraint over cardinality (TCOC): A given adaptation action can be executed successively at most \( N \) times within a time period \( T \). There are two steps in order to add a TCOC constraint to our Event-B specification. First,
the number of times to execute a given adaptation action is recorded in a variable \( Nbex \) (Act2). Then, in the event execute, a guard (Grd2) is needed, which forces the event to be eligible to occur only if the number of times \( Nbex \) is less than \( N \) and the current time is less than the time period \( T \). A general pattern specifying the TCOC property is shown below:

**Stabilisation Time:** When a node executes an adaptation action, it is labelled an unstable node. After a reconfiguration time (ExDuration), a timer with a certain stabilisation time (ST) is set. When this time expires, the unstable state will change to the stable state and a new adaptation control loop can begin with the monitoring event.

Let \( Te \) be the starting time of the event Execute. Let \( ST \) be the time required for the system to stabilise. Let \( ExDuration \) be the required time for executing an adaptation action. The stabilisation time constraint is defined in the event Execute as:

\[
\text{Time} > Te + ExDuration + ST.
\]

The specification of the stabilisation time property is presented as follows:

By applying transformation rules on the generated XML specifications of the master/slave pattern applied to the FFDS, we obtain Event-B specifications presented in Figure 9. The resulting specifications are enriched with adaptation, system and temporal properties. By using the Master/Slave pattern, all the proof obligations of machines M0, M1 and M2 are proved automatically as shown in Figure 9, since they are correct-by-construction.
8 Tool support

Expiry time: A given event cannot occur if the expiry period has passed. For instance, if the planning event receives a request for change when the expiry time is achieved, this request will not be considered. As shown below, in order to specify an expiry time for an event, an action (Act1) is needed to record the occurrence time \( T_a \) in the trigger event (event Analysis), and a guard (Grd2) in the restricted event (event Planning) to prevent it from happening if the expiry period \( E \) has passed.

To achieve our purpose, we propose an Eclipse plug-in whose global architecture is shown in Figure 2. It will help us to implement our approach in a practical context. We propose a graphical modelling tool that makes the modelling of self-adaptive systems easier. It is based on the Frameworks GMF (Graphical Modelling Framework), EMF (Eclipse Modelling Framework) and GEF (Graphical Editing Framework). Five MAPE patterns are available in our Eclipse plug-in: master/slave, regional planner, hierarchical control, coordinated control and information sharing.

In our editor, graphical elements can be picked up from a tool palette and created in the diagram editor pane in a drag-and-drop way. Elements of the palette are listed under Nodes, MAPE components, MAPE sub-components, Dependency and Link elements. Figure 10 shows the tool palette of the master/slave pattern.

Figure 10 Tool palette

After completing the modelling phase of a self-adaptive system, our plug-in takes as input the generated XML file describing the designed self-adaptive system. Then, it transforms the generated XML file, according to transformation rules expressed with the XSLT language, into Event-B specifications.

Thirdly, these specifications are imported into the Rodin platform to automatically check a set of properties which are already provided. Therefore, via this tool we ensure the automatic generation of a file which contains the different properties that will be verified later. They differ from one another and fall into three categories as previously mentioned.

9 Related work

Over the last decade, a number of research studies have been conducted in an attempt to model, specify and verify self-adaptive systems (Güdemann et al., 2006; Cardozo et al., 2013; Weyns et al., 2012; Magee and Maibaum, 2006). We start the discussion of the related work section with a selection of research approaches on modelling self-adaptive system (SAS). Then, we focus on the studies that propose formal specification and verification of self-adaptive systems.

9.1 Modelling self-adaptive systems

Hebig et al. (2010) proposed a UML profile that extends UML component diagram used for modelling two use cases: a communication subsystem of a mobile agent of a self-organising system and autonomous train vehicles. In line with our work, Hebig et al. (2010) support the designing of multiple control loops and their interplay at the architectural level. However, this work focuses only on modelling structural features of self-adaptive systems and does not provide detailed individual steps of the adaptation process.

Vogel and Giese (2012) propose runtime mega-models to ease the development of the adaptation logic by providing a domain specific modelling approach and a runtime interpreter for self-adaptive system. This supports development by explicitly modelling the feedback loop at a higher level of abstraction. Their work has considered detailed individual adaptation steps, like monitoring, analysis, planning and execution. However, the authors in this work did not consider structural features of self-adaptive systems and did not provide a tool support to assist the designer in modelling complex self-adaptive systems.

Luckey and Engels (2013) used a concrete UML-based CSML (concern specific modelling language) named ACML (Adapt Case Modelling Language) to model structural and behavioural features of self-adaptive software systems. It is heavily UML based. Besides, they use semantic specification language for dynamic meta-modelling (DMM) to enable the quality assurance of self-adaptive systems using model checking technique.
Abeywickrama et al. (2013) proposed the SOTA (State Of The Affairs) approach that supports an early, goal-level, model checking analysis for adaptive systems. However, the authors of this work adopted a very complex model checking process involving several formalisms: the i* framework is used to model the static aspects. An operational SOTA language is defined and used to describe the dynamic aspects and dependencies among different components. LTSA (Labelled Transition System Analyser) is then provided to formally define the goal or utility for verification purposes.

Indeed, the existing approaches dealing with designing self-adaptive systems do not propose a tool support to assist the designer in modelling structural and behavioural features of self-adaptive systems with multiple control loops. Moreover, little attention has been paid to provide a set of MAPE patterns for self-adaptive systems. The main goal behind this is to facilitate application-independent instantiation of models for complex self-adaptive systems. Furthermore, using design patterns in modelling self-adaptive systems leads to a reduction of development timescales.

### 9.2 Formal specification and verification of self-adaptive systems

Providing evidence that self-adaptation properties are satisfied has long been a topic of intensive research (Luckey and Engels, 2013; Weyns et al., 2010; De La Iglesia and Weyns, 2015; Arcaini et al., 2015; Camilli et al., 2015). Luckey and Engels (2013) used a concrete UML-based CSML (concern specific modelling language) named ACML (Adapt Case Modelling Language) to model structural and behavioural concerns of self-adaptive software systems. These models are then formally defined using a temporal logic and taken as input for a model checker. The model checker result is used to create a quality analysis report (absence of deadlocks, stability, etc.) which is provided to the system designer during design-time. However, this work does not support the verification of complex self-adaptive systems with multiple control loops and does not support the verification of temporal properties.

Weyns et al. (2010) proposed FORMS (a formal reference model for self-adaptation). It is a reference model for self-adaptive systems built on established principles of self-adaptation, such as architecture-based self-adaptation, computational reflection, and MAPE-K. However, FORMS suffers from the over-specification due to the rigidity of the Z formalism and does not allow the analysis of different properties for self-adaptive systems to ensure the correctness of the system adaptation logic.

De La Iglesia and Weyns (2015) suggested extending the existing mobile learning application with a self-adaptation layer, making the system robust to the degrading GPS accuracy. The self-adaptation layer is conceived as a set of interacting MAPE loops distributed over the mobile devices. To guarantee the robustness requirements, they formally specified the self-adaptive behaviours using timed automata, and the required properties using TCTL (Timed Computation Tree Logic). Besides, they used the Uppaal tool to model a self-adaptive system and verify the robustness requirements. However, the verification proposed on this work is specific to the functionality of the proposed use case.

Arcaini et al. (2015) proposed modelling and analysing MAPE-K loop for self-adaptation using the concept of multi-agent ASM (Abstract State Machines). They proposed a verification technique of different properties (flexibility and robustness) through model checking, to discover unwanted interference between the MAPE loops. Indeed, the model checking technique is computationally expensive as it suffers from state explosion problem especially for large and complex systems. Besides, Arcaini et al. (2015) did not verify temporal properties for self-adaptive systems.

Camilli et al. (2015) proposed a formal approach to specify and verify the self-adaptive behaviour of real-time systems. Their specification formalism is based on Time-Basic Petri nets, a particular timed extension of Petri nets. They proposed the verification of inter-zone properties to check the correctness of the behaviour of the entire system. Additionally, this research proposed the verification of intra-zone properties to check the correctness of the behaviour of a single zone of interest, such as invariant, safety and liveness. They used the GRAPHGEN software tool as a simulation technique to verify the correctness of properties. Due to the complexity of such systems, simulation and testing alone are no longer sufficient to gain a sufficiently high quality of the design. Moreover, in critical and real time applications, it is mandatory to formally prove that the system meets the given specifications.

Most of the existing research approaches that proposed formal verification of self-adaptive systems used the model checking technique. However, this latter is known to be computationally expensive, as it suffers from the state space explosion problem. Any solution based on such technique, either used at design time or run-time is restricted with respect to providing guarantees in terms of system size.

### Table 1 Summary table of related work on SAS modelling

<table>
<thead>
<tr>
<th>Approach</th>
<th>Tool support</th>
<th>SAS modelling language</th>
<th>Reutilisability</th>
<th>Structural and behavioural modelling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hebig et al. (2010)</td>
<td>–</td>
<td>UML profile</td>
<td>–</td>
<td>structural</td>
</tr>
<tr>
<td>Vogel and Giese (2012)</td>
<td>–</td>
<td>Runtime megamodel</td>
<td>–</td>
<td>behavioural</td>
</tr>
<tr>
<td>Luckey and Engels (2013)</td>
<td>–</td>
<td>ACML</td>
<td>–</td>
<td>structural and behavioural</td>
</tr>
<tr>
<td>Abeywickrama et al. (2013)</td>
<td>–</td>
<td>UML profile</td>
<td>+</td>
<td>behavioural</td>
</tr>
<tr>
<td>Our approach</td>
<td>+ (Eclipse plug-in)</td>
<td>UML profile</td>
<td>+</td>
<td>structural and behavioural</td>
</tr>
</tbody>
</table>
As argued earlier, existing research approaches either do not target the verification of temporal constraint or only propose a verification of a limited set of temporal constraints.

Then, the main contribution of our work is to address the problem of modelling and verifying self-adaptive systems with multiple MAPE control loops. The proposed solution takes into account structural and behavioural features. Besides, our refinement based approach will help the architect to design step-by-step a self-adaptive system instantiating a MAPE pattern. We use the Event-B method as a formal notation to verify a set of behavioural properties for self-adaptive system. Moreover, we propose the verification of interesting timed properties to check that adaptation comply with the specific temporal deadlines.

10 Conclusion and future work

In this paper, we have presented a refinement based approach for modelling, specifying and verifying complex self-adaptive systems which instantiate MAPE architectural patterns. We have proposed a UML profile to represent the self-adaptive systems which instantiate MAPE architectural approach for modelling, specifying and verifying complex systems.

Our approach

Event-B method
Simulation (GRAPHEN)
Theorem proving (Rodin)

References


Table 2 Summary table of related work on SAS verification

<table>
<thead>
<tr>
<th>Approach</th>
<th>SAS specification language</th>
<th>Used technique</th>
<th>Temporal properties verification</th>
<th>Support of SAS with multiple control loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luckey and Engels (2013)</td>
<td>Temporal logic</td>
<td>Model checking (Groove)</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>De La Iglesia and Weyns (2015)</td>
<td>Timed automata</td>
<td>Model checking (UPAAL)</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Arcaini et al. (2015)</td>
<td>CTL and LTL</td>
<td>Model checking (NuSMV)</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>Camilli et al. (2015)</td>
<td>Time basic petri nets</td>
<td>Simulation (GRAPHEN)</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Our approach</td>
<td>Event-B method</td>
<td>Theorem proving (Rodin)</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>


