Specifying and Verifying the Context-aware Adaptive Behaviour of Software Systems

Technical Report: C3-516_03

Mahmoud Hussein, Jun Han, and Alan Colman
Faculty of Information and Communication Technologies (FICT)
Swinburne University of Technology, Australia

December 2010
# Table of Contents

TABLE OF CONTENTS

1 INTRODUCTION

2 MOTIVATING SCENARIO AND REQUIREMENTS ANALYSIS

3 SPECIFYING AND VERIFYING THE SYSTEM ADAPTIVE BEHAVIOR
   3.1 SPECIFYING THE SYSTEM CONTEXT-AWARE ADAPTIVE BEHAVIOR
   3.2 VERIFYING THE SYSTEM ADAPTIVE BEHAVIOUR
       3.2.1 Building the System Adaptive Behaviour State-based Model
       3.2.2 The System Adaptive Behaviour Errors
   3.3 SPECIFYING AND VERIFYING THE SYSTEM ADAPTIVE BEHAVIOUR USING OUR TOOL

4 CASE STUDY
   4.1 THE CONTEXT-AWARE ADAPTIVE BEHAVIOUR OF THE VEHICLE ROUTE PLANNING SYSTEM
   4.2 VALIDATING THE CONTEXT-AWARE ADAPTIVE BEHAVIOUR VISUALLY
   4.3 VERIFYING THE ADAPTIVE BEHAVIOR USING ROMEO TOOL

5 RELATED WORK AND DISCUSSIONS

6 CONCLUSIONS AND FUTURE WORK

REFERENCES
Faculty of Information and Communication Technologies  
Swinburne University of Technology

Specifying and Verifying the Context-aware Adaptive Behaviour of Software Systems

Mahmoud Hussein, Jun Han, and Alan Colman

Abstract

Context-aware adaptive software systems need to have a model for the system adaptive behaviour. This model decides the system reactions in response to the environment changes. In large scale software systems with high variability, an explosion in the number of system’s states (i.e. configurations or behaviours) and the transitions between them (i.e. the system adaptive behaviour) is introduced. Therefore, specifying the system adaptive behaviour and assuring its correctness are major challenges. Existing research solves only one of these two challenges where improving one aspect makes the other worse. In this report, we introduce a novel approach to specifying and verifying the context-aware adaptive behaviour of a software system. Our approach explicitly represents the relationships between the context changes and the system variations, so that the system adaptive behaviour is easily captured. We classify the possible system variations into dependent and independent variations for reducing the possible system states and the transition between them. The designed system adaptive behaviour model is transform to a Petri Net model so that it can be verified for detecting the adaptation behaviours errors such as inconsistency, redundancy, circularity, and incompleteness. We generate the system adaptive behaviour implementation from its model. In addition, we illustrate our approach though specifying and verifying the context-aware adaptive behaviour of the vehicle route planning system.

Keywords: Context-awareness, self-adaptivity, adaptive behaviour model, verification, model-based development, formal specification.

1 Introduction

There is an increasing demand for software systems that dynamically adapt their behavior at run-time in response to changes in their requirements, users’ preferences, operational environments, and underlying infrastructure [1-2]. Changes can also be induced by failures or unavailability of parts of a software system itself [3]. In these circumstances, it is necessary for a software system to change itself as necessary to continue achieving or preserving its new and existing goals. A challenge is how to specify, design, verify, and realize such systems that evolve at runtime [1-5].

The context-aware adaptive software systems need to have a mechanism that decides the system reactions in response to the context changes (i.e. the system adaptive behavior model). This mechanism need to be built carefully, where the triggering of incorrect adaptation actions or the correct one is not triggered leads to undesired system behavior (i.e. the system behaves improbably) [6-7]. The larger the number of environment variables that need to be taken into account, the complex the system adaptive behavior will be. In addition, multiple adaptations actions can be triggered concurrently where the environment variables are shared between the conditions that trigger the
system adaptation. Therefore, the specification and the analysis of the system adaptive behavior model are challenging tasks. Current mechanisms to decide the required adaptation actions in response to context changes are classified into three categories: (1) rule-based mechanism, where it takes the form of “IF (condition, i.e. the context change), THEN (actions, i.e. the required adaptation) [8-9], (2) goal-based mechanism, where the system goals are defined and used to infer the required adaptations actions rather than specifying what to do in the current context situation such as the rule-based mechanism [6, 10]; (3) utility-based mechanism, where it generalizes the goal based mechanism by quantifying each possible state with a value instead of classifying the states as desired next state or not (i.e. 0 or 1) [11-12]. On the one hand, rule-based approaches are expressive and easy to write, but are prone to error (e.g. rules conflict, redundancy, etc.) and need to be verified. On the other hand, the goal-based and the utility-based mechanisms are difficult to build and have the state explosion problem in large scale systems, but they can be verified for correctness [13]. As such, there is a need to have an approach that combines the expressiveness and ease in building of rule-based mechanism and the formalism of the state-based mechanism to enable its verification.

In this report, we introduce our approach for specifying and verifying the adaptive behavior of a software system. It has the expressiveness/easiness of the rule-based mechanism and the formality of the state-based model. In addition, we provide a graphical representation of the system adaptive behavior model by representing explicitly the relationship between the context changes and the system adaptation for preserving and/or achieving the system goals. Compared to existing work, our approach has the following novel features. First, our approach represents the relationships between the context changes and the system variations (i.e. the system configurations/behaviours) explicitly, so that the system adaptive behaviour is easily captured. Second, we classify the possible system variations into dependent and independent variations for reducing the possible system states and the transition between them (i.e. the state explosion problem cannot happen easily). From the system reconfiguration point of view, independent variations means the change of a system component is not dependent on the other components while dependent variations means the change of a component is depended into another (e.g. the replacement of a realization for a component that can be added and removed). Finally, the designed system adaptive behaviour model is transform to a Petri Net model so that it can be verified for detecting the adaptation behaviours errors. These errors are (a) inconsistency, where a conflicting adaptation actions are fired concurrently (e.g. adding and removing the same system component); (b) redundancy, where the same adaptation action is fired twice in the same context state; (c) circularity, where the chain of fired adaptation actions are keep repeating (i.e. an endless loop with the decision making mechanism); (d) incompleteness, where a missing adaptive behaviour is detected by identifying a context situation that has no reaction from the system.

The remainder of the report is organized as follows. We start by introducing a motivating scenario in section two. Our adaptive behavior model specification and verification approach is given in section three. In section four, we demonstrate our approach and tool through specifying and verifying the context-aware adaptive behavior of the route planning system. Section five analyzes existing work with respect to our approach. Finally, we conclude the report in section six.

2 Motivating Scenario and Requirements Analysis

The vehicle route planning software system helps the driver to plan his journey by providing suitable routes from the current location to a destination. Below are a few scenarios where this system takes into account different sources of context information in dynamically planning the travel route.
The context information may include (a) the dynamic traffic information available from roadside units or a traffic information service provider (through Wi-Fi, DSRC and/or 3G technologies), and (b) the driver preferences (such as shortest or fastest or most carbon-efficient route) from his mobile phone. However, vehicles come in different models and with different enabling technologies. For example, some vehicles do not have the ability to communicate with the driver’s mobile to get his preferences. The traffic information may or may not be available depending on the communication technologies installed and the availability of the roadside units or traffic information provider. Therefore, the vehicle route planning system should use different variants (i.e. algorithms) to compute the possible routes based on the available context information.

When the vehicle is running in high speed such as over 70 km/h, it is difficult for the driver to concentrate on both the road and the displayed route map. In this situation, the system should use voice instruction for the vehicle route to reduce driver distraction. As such, the system should add the voice instruction component when the vehicle is moving in high speed and the driver is not using the voice instruction option.

The above scenarios show a number of general requirements. First, the system should adapt itself (i.e. adaptive) in response to context changes. For example, adding the voice instruction component when the vehicle is moving with high speed and ordering the displayed routes based on the driver route preferences. As such, the system should have a mechanism that decides the system reactions/adaptations in response to the context changes (i.e. an adaptive behavior model). Second, the system’s environment has a large amount of information about the driver, the vehicle, and the vehicle environment (e.g. the nearby vehicles, the services providers, the roadside units, etc.), which affects the system operation and need to be taken into account. When there are a large number of environment variables that need to be taken into account, the system adaptive behavior become complex and its design is error-prone. Therefore, the mechanism used to specify the system adaptive behavior should be able to (a) tackle the large number of environment variables and the system reactions to them (i.e. easy to be built) and (b) verify the adaptive behavior to avoid the adaptation errors that lead to undesired system state while the system is in operation.

3 Specifying and Verifying the System Adaptive Behavior

The system needs to detect its context information changes and adapt upon them, and then there is a system adaptive behaviour model captures the relationship between the context changes and their corresponding system reactions. To reduce the complexity of modelling the system adaptive behaviour in large scale software systems, we separate the context model from the system adaptive behaviour model and capture their relationships explicitly using our component model as discussed in the next sub-section.

3.1 Specifying the System Context-aware Adaptive Behavior

For modelling the system adaptive behaviour, we introduced a component model (shown in Figure 1) that specifies explicitly the required and provided context and adaptation action ports, and the adaptation condition [14]. First, the adaptation mechanism requires the environment states that are provided by the context providers to make the adaptation decisions. Consequently, we explicitly reflect the requirement and provision of such context information in our component model through the required and provided context information ports. Second, the system adaptive behaviour model is used to decide the required adaptation actions, and then it should explicitly define these required adaptation actions. In addition, actual adaptation actions need to be performed on the relevant components that should specify explicitly what adaptation actions they support. For example, the
route planner component has the ability to switch between different route planning realizations. As such, our component model has explicit required and provided adaptation action ports as shown in Figure 1. Third, we add the *enabling condition* element to our component model (see Figure 1) for the adaptation *condition definition* (e.g. vehicle speed is greater than 70 km/h). Finally, our component model contains the traditional required and provided function ports for representing the system core functionality.

Figure 1: Our context-aware adaptive systems component model

In Figure 2, we show an example of the system adaptive behavior using our component model. First, the whole adaptive behavior model is represented as a composite component (i.e. the change management), which have sub-components to represent the system reactions to the context changes. Each sub-component (e.g. $R_2$) has the enabling rule condition(s) (e.g. $\text{Attribute}_2 > 40$), and the rule actions as a required adaptation action ports (e.g. Add Component2). In addition, the context attributes in the rule conditions are exposed as required context ports of the rule component to obtain their values from the context providers (e.g. Attribute 2). Furthermore, this composite also has some provided adaptation action ports to enable its own adaptation (e.g. Remove R2) if required. The following are the three rules represented in Figure 2:

$R_1$: IF the component one state is active, THEN the system removes the context entity two.

$R_2$: IF the context attribute two value is greater than 40, THEN the system adds component two.

$R_3$: IF the context entity two is active, THEN the system removes the adaptation rule two.

Figure 2: Specifying the context-aware adaptive behaviour using our component model

Modelling the system adaptive behaviour using the above mechanism has the following advantages. First, it is an easy method to capture the relationship between the context changes (i.e. the required context information and the enabling condition on it) and the system adaptation/reaction (i.e. the required adaptation action port). Second, the context information processing and management is separated from the system adaptive behaviour model and then the system modelling complexity is reduced. Third, modelling the system adaptive behaviour as condition-action rules enables a fast system reaction during the runtime where the system specific reactions are specified during the design time. Forth, we represent the system adaptive behaviour as components that have a runtime representation, and then it can be changed at runtime to incorporate a new adaptive behaviour. Finally, we do not need to enumerate all the system states to build the adaptive behaviour model such as the goal-based and utility-based mechanisms.
3.2 Verifying the System Adaptive Behaviour

For verifying the system adaptive behaviour using the existing model checkers (e.g. Romeo tool [15]), we need to (a) transform the specified adaptation rules using our approach to a state-based model (e.g. Petri Net [16]) and (b) specify the properties the generated model need to be checked against.

3.2.1 Building the System Adaptive Behaviour State-based Model

The first step to build a state-based model is to enumerate its states, which corresponding to the possible variations (i.e. configurations and behaviours) the system can be in during the runtime. Form the system re-configurations point of view\(^1\), the number of the system states can be calculated as the product of the possible variations of the system components. For example, if we have a system that is consisted of 10 components. First, two components can be added and removed as needed. Second, another two components have three variants which one of these variants is selected based on the system requirements. Finally, the other six components are fixed which represents the system basic functionalities. The number of possible variations of the system is 36 states (i.e. \(2\times2\times3\times3\)). Furthermore, if we considered a component from the six fixed components has three variants, then the total number of system states is increased to 108 states (i.e. \(3\times36\)). Therefore, the number of the system states grows exponentially with the system possible adaptations, and then there is a need for a method to reduce the number of these states.

In our approach, we reduced the number of the system states (i.e. to avoid the state explosion problem) by considering the system state as a combination of multiple sub-states of the system components variations (i.e. not the whole system configuration as one state). In addition, for taking into account the components dependencies, we classify the possible system variations into two types: (1) independent variations, where the change of a system component is not dependent on the other components; (2) dependent variations, where the change of component is depended into another. For example, the selection of a component realization is dependent on the component availability (i.e. is the component enabled or not). The for each dependent variation group, a state model is constructed, which have number of states equals to the product of this group members variations. Finally, state models are constructed for each independent system component variation (i.e. model of two states in case of adding/removing a component or model of n states where n is the number of the component possible variants/implementations).

Figure 3 shows a system that has two components. Each component has three variations (i.e. realizations one, two, and three). In addition, the component two can be added and removed as required. Following the traditional approach in enumerating the possible system state, the system can have 12 states (i.e. the different combinations of applying the system adaptation actions). In Figure 3, we show only four variations (states) of the system. Variation one (i.e. state one) has the two components active, and the realization one for the two components is selected. When the system removes the component two, the result is configuration two (i.e. state two). States three and four show the system when it keeps the realization of component one as it is and changes the realization of component two from one to two and then three. The other eight system states are the same as these four states except the change of component one realization, where there are four states when the component one realization two is selected and the other four state when the realization three is selected for component one.

\(^1\) In the following, our discussion is more concerned with the system reconfiguration as adaptation actions. The system parameters and behavior changes are treated by the same manner.
In our approach, we do not consider the whole configuration as state where we divide each state to sub-states that is corresponding to the single component adaptations. Therefore, we have three states for component one (i.e. the component different realizations), and four states for the component two additional and removal and the selection for its realization where they are dependent on each other as shown in Figure 4. The system state is the combination of states of component one and two. For example, state one in Figure 3 is corresponding to the combination of state one in Figure (4-A) and state two in Figure (4-B).

In addition to the reduction of the states from 12 to 7, the transition between these states (the system adaptive behaviour) is also reduced. The number of the transitions between n states equals n-1 plus the number of transition between n-1 states. Therefore, there are 66 transitions with the model of 12 states using the traditional approach. In our approach, we have 3 transitions within the model of component one states and 6 within the component two state model (i.e. a total of 9 transitions). In the next subsection, we follow the above approach to generate the Petri Net model and use it for detecting the adaptation behaviour errors.

### 3.2.2 The System Adaptive Behaviour Errors

In large scale software system where there are a large number of adaptations, the system adaptive behaviour is subject to errors such as **inconsistency, redundancy, cycles, and incompleteness**. As such in the following we present (a) the definition for each error type; (b) an example that shows how it can happen with the example corresponding Petri-Net to enable the error detection via Romeo tool.

#### A. Adaptation Behaviors Inconsistency

The inconsistency means that the adaptation actions that need to be applied into the system are contradicting each other. The system possible adaptation actions are adding, removing, and replacing a system element. The inconsistency between these actions can happen in the following situations.
First, the required adaptation actions are to add and remove the same system element (type 1 error). Second, the required adaptation is to change (i.e. replace) the system element twice (type 2 error). For example, there are two replacements actions of the same component in the adaptation script (e.g. replace component 1 with component 2 and replace component 1 by component 3). In the following, we present a set of adaptation rules contains the above errors and how these errors can be detected.

**Adaptation rules that have type 1 error**

*R1*: If the vehicle speed is greater than 40 km/h, then the system **adds** the voice instruction.

*R2*: If the vehicle speed is lower than 50 km/h, then the system **removes** the voice instruction.

In Figure 5, we transform the above rules to a Petri Net model (shown in Figure 5-A), where the rules conditions are represented as input places (e.g. VehicleSpeed_GreaterThan_40), and the rules adaptation actions as output places (e.g. Add_VoiceInst). Each rule is captured using the Petri Net transition (e.g. R1) that links between the input and output places, and then the evaluation of the rules condition to true actives the rules action(s). The input place that represents the rule condition can be shared between multiple adaptation rules, and then we consider it as output place too where the transition keeps the input place active to be used in other adaptation rules. But the adaptation rules can be activated again, and then we added the rule enabling condition for making the rule evaluation to true once (e.g. R1_Enabling). This way of transforming the adaptation rules to Petri Net follows what was described above. We put each single adaptation action into an output place, and then the whole system configuration (i.e. the system state) can be inferred using multiple output places (i.e. the sub-states we mentioned above for reducing the state space) and not a single place such as the traditional approach [6, 10].

To verify the inconsistency between these two rules using the generated Petri Net, we used Romeo tool [15]. The inconsistency can be checked visually by playing the token games and then looking for a state where both rules one and two are evaluated to true. If we considered the vehicle speed is equal to 45 (i.e. the initial marking of the Petri Net in Figure 5-A), then rules one and two are activated (i.e. the final marking of the Petri Net as shown in Figure 5-B). For the verification using the Romeo tool model checker, we represented the conflict type 1 error property using Timed Computation Tree Logic (TCTL) [17] as “EF[0,0](M(3)+M(6))>1” (shown in Figure 5-C). This formula means that there exist a path where the marking of the Petri Net places 3 (i.e. Add_VoiceInst) and 6 (i.e. Remove_VoiceInst) are greater than 1 (i.e. both actions are active). The bottom of Figure 5-C shows that this formula is evaluated to true when R1 and R2 are activated (i.e. there is a conflict between rules 1 and 2, where the two adaptation actions are contradicting to each other).

**Adaptation rules that have type 2 error**

*R3*: If the rain level is heavy, then the system sets the **vehicle speed limit** to 50 km/h.

*R4*: If the temperature is greater than 40, then the system sets the **vehicle speed limit** to 90 km/h.

---

![Figure 5: Petri Net for representing adaptation rules one and two and their verification](image_url)
R5: If the driver preference is available, then the system uses the route planning one.
R6: If the congestion information is available, then the system uses the route planning two.

A Petri Net for rules three and four is shown in Figure 6-A. When the rain level is heavy and the temperature value is over 40 degrees, the two rules are evaluated to true and then the two adaptation actions are fired. These two actions overwrite each other, and then they are inconsistent. To verify this inconsistency, the following TCTL formula can be used “EF[0,0]M(p1)+M(p2)>1” which means the output places p1 and p2 are active in the same time (e.g. places Set_SpeedLimit_50 and Set_SpeedLimit_90 in Figure 6-A). Similarly, the adaptation rules five and six are used to select the suitable route planning algorithm. When both conditions are true, the two adaptation actions are fire together as shown in Figure 6-B and they are inconsistent adaptations.

Figure 6: Petri Nets for representing adaptation rules three to six

B. Adaptation Behaviours Redundancy

The redundancy appears when a rule is repeated as it is, or one rule is a sub-part of another. For example, two rules have the same condition(s), and the adaptation action(s) of a rule is a part of the other rule adaptation action(s) (type 3 error). This error is detected by looking for an adaptation action that is repeated twice in the required adaptation actions.

Adaptation rules that have type 3 error
R1: If the vehicle speed is greater than 40 km/h, then the system adds the voice instruction.
R2: If the vehicle speed is greater than 40 km/h, then the system adds the voice instruction component and uses the route planning algorithm one.

Figure 7 shows the Petri-Net state, where the two rules are evaluated to true. The result is an adaptation script that has Add_VoiceInst adaptation action appears twice. This error can be detected by checking the Petri Net using the EF[0,0]M(3)>1 property. This means that output place three activated twice (i.e. Add_VoiceInst). The verification shows that the evaluation of the adaptation rules one and two satisfies this property (i.e. the right part of Figure 7).

Figure 7: Petri Net for representing adaptation rules that has the redundancy problem

C. Adaptation Behaviours Cycles

In the context-aware adaptive systems, the context model changes makes the system adapts (e.g. when the context model has the driver preferences entity active, the route planning algorithm one is
selected). In addition, the functional system changes can lead to a context model changes (e.g. in response to the driver selection to use the route planning two, the context model is changed by activating the congestion information context entity). Therefore, the adaptation rules for changing the functional system in response to the context model changes and vice versa should be written carefully to avoid the cycles. The cycle happen when the adaptation rules evaluation leads to adaptation actions that make the same chain of rules firing to be performed again (i.e. type 4 error).

**Adaptation rules that have type 4 error**

R1: If the driver preference is active, then the system activates the route planning one.
R2: If the route planning one is active, then the system activates the driver preference.

Figure 8-A shows the Petri-Net where the rule one condition is active. After firing rules one and two (Figure 8-B), the rule one condition is activated again and this action is unwanted. Therefore, there is a cycle between these two rules, and then the system keep going back and forth between them. Figure 8-C shows how this error is detected by checking the model against $EF[0,0]M(1)>1$ formula. The checking of this property to true means that the adaptation rules actions (i.e. R1 and R2) enable the condition of rules one again which in turn cause rules one and two to be fired continuously.

![Figure 8: Petri Net for representing adaptation rules that has cycles and how they are detected](image)

**D. Adaptation Behaviours Incompleteness**

In large scale systems, there are a large number of adaptation behaviours. As consequence, there is a possibility of having missing adaptation behaviours (type 5 error). These missing behaviours are appeared when there is a context situation without having an adaptation action to it or the rule conditions cannot be evaluated to true (i.e. the rule cannot be fired). For example, an adaptation rule is based on an and-condition (e.g. A and B), but the condition A and B cannot be evaluated to true in the same time.

**Adaptation rules that have type 5 error**

R1: If the driver preference is active and the congestion information is not active, then the system activates the route planning one.
R2: If the congestion information is active and the driver preference is not active, then the system activates the route planning two.
In Figure 9, we show the Petri Net that represents adaptation rules one and two. We show three different marking for the net. First, the rule one condition is true (Figure 9-A). Second, the condition of rule two is true (Figure 9-B). Third a part of both rules one and two conditions is true (Figure 9-C). We defined the completeness property as “EF[0,0]M(4)+M(8)>0”, which means that at least one of the two rules is evaluated to true. The initial marking of Figure 9-A and B satisfies this property where rule one or two is fired. However, Figure 9-C do not satisfy this property, where there is no rule is fired when both the congestion and driver preferences are active (i.e. missing adaptation behaviour).

3.3 Specifying and Verifying the System Adaptive Behaviour using our Tool

To support specifying and verifying the context-aware adaptive behaviour of a software system using our approach, we extended our context-aware adaptive systems development tool (CAST) [18]. This extension is to enable the software engineer to model, generate the implementation, and validate/verify the system context-aware adaptive behaviour. The process outlines our approach and its tool support is shown in Figure 10 and it is as follows:

Step 1: The software engineering use our component model to specify the system adaptive behaviour in the rule-based form (e.g. Figure 2), and this model is the input to our tool [14].

Step 2: Using our tool the software engineer can (1) visually test the adaptive behaviour of the system as will be shown in next section. In case of there are some errors detected, the adaptive behaviour model is changed and the steps one and two are repeated; (2) generate a Petri Net model that is corresponding to the rule-based model as an XML file.

Step 3: The generated Petri Net model is passed to the Romeo tool. This tool verify the system adaptive behaviour, and in case of errors detection, the software engineer go back to modify the rule-based model.
based models, and repeats the steps one through three again until having a model that represent the adaptive behaviour correctly. When he become satisfied to the create model, the implementation corresponding to the adaptive behaviour model is generated automatically.

We extended our tool to have the ability to specify and transform the rule-based model of the system adaptive behaviour to a Petri Net model. Using our tool GUI, an adaptation rule can be specified as shown in Figure 11-A. This rule has a condition “Vehicle_Speed > 40” and adaptation action “Add_VoiceInst”. In our model, we specify the adaptation rule as component, and then an XML description of R1 component is shown in Figure 11-B.

After specifying the adaptation rules using our tool, the tool is used to generate the Petri Net model in the form of XML file. This XML file is the input to the Romeo tool. The generated XML file that is corresponding to the above rule is shown in Figure 12. Figure 12-A shows the places that are used to specify the rule condition (i.e. place 1), action (i.e. place 3), and enabling (i.e. place 2). The transition that link these places together is shown in Figure 12-B. In Figure 12-C, the arcs that links the input/output places with the transition are shown. The arc between the input place and the transition has the type “PlaceTransition”, and the arc between output place and transition has the type “TransitionPlace”. Then, the Romeo tool can visually validate the net behaviour by playing the tokens games or use TCTL model checker to verify this model according the properties specified in previous sub-section (e.g. inconsistency, redundancy, etc.).
In this section, we use the steps described above to specify and verify the context-aware adaptive behaviour of the vehicle route planning system (described in section two). We present the system adaptive behaviour model in sub-section one, the visual validation of it is discussed in sub-section two, and then the adaptive behaviour verification using Romeo tool is described in sub-section three.

4.1 The Context-aware Adaptive Behaviour of the Vehicle Route Planning System

The system adaptive behaviour captures the relationship between the context changes and the system reactions, and then the modelling of the system adaptive behaviour is not separate from the context and the functional system modelling. In Figure 13, we show the system model that includes the context model, the functional system model, and the adaptive behaviour model using our proposed component model [14]. In this example, we designed the adaptive behaviour to highlight the possible errors discussed in the previous section. This model has the following elements.

**The Context Model:** The context model has three entities (components): the vehicle information, the driver preferences, and the traffic information as shown in Figure 13. These entities represent the environment information that is needed by the route planning system to continue its operation or for triggering the system adaptation. In addition, the context composite component is able to add, remove, or replace the context entities (e.g., remove the traffic information entity). Furthermore, the providers for this context information are: (a) the On-Board Diagnostic (OBD) system for providing the vehicle speed; (b) the driver’s mobile for providing his route preferences; (c) the traffic information service provider and road side units for providing the traffic congestion information; (d) the traffic information and the driver preference context entities for providing their availability.

**Functional system model:** It represents the system functionality and has two components as shown in Figure 13. First, the route planner provides the possible routes between the current location and destination. These routes are computed by different algorithms based on the available context information. The route planner component has the ability to switch among these different route planning algorithms implementations. For example, route planning two is used when the traffic information and the driver preferences are both available. Second, the route planning display presents to the driver the route computed by the route planner onto a map together with the journey progress and voice instructions. There are two variants for this component: (a) only the map with journey progress information over it using only the map component; (b) the map with the journey progress and...
voice instructions for the selected route using both the map and the voice instruction components. This variation is achieved by adding and removing the voice instruction component.

The first component is realized in Figure 13 through a three different algorithms for the route planner. The default route planning component takes the vehicle current location and the destination and provides the possible routes without taking into account any context information. The route planning one component considers the driver route preferences in calculating the routes. In addition, its state (i.e. the component is selected and used by the system or not) is provided by route planning one monitor. The component route planning two provides the available routes based on both the traffic congestion information and the driver route preferences. Besides, there are realizations for displaying the computed route onto a map and for providing the voice instructions for the selected vehicle route for realizing the second component.

![Figure 13: Context-aware adaptive vehicle routing planning system](image)

The Adaptive Behaviour Model: The change management composite component (see Figure 14) consists of a set of rules that are used to determine the required adaptation actions in response to the context changes. Our example has many adaptation rules. We show only six adaptation rules that contains the different adaptation behaviour errors discussed above.

1. When the driver uses route planning one, the system needs to consider the driver route preference only, and then the traffic information context entity needs to be disabled to reduce the monitoring overhead. As such, the component rule one (R1) in Figure 14 has the enabling condition “is the driver uses route planning one (i.e. active)?” and the required adaptation action “remove traffic information” context entity.
(2) The traffic information provider can be disabled due to the communication link problems during the vehicle journey, and then we defined the adaptation rule two (R2). This rule makes the system switches to using the route planning one (i.e. the required adaptation action), when the traffic information is not available (i.e. the rule enabling condition).

(3) The availability of the driver route preferences enables the selection of the route planning one. Therefore, the component rule three (R3 in Figure 14) defines the availability of route preference as the rule enabling condition and the use of route planning one as the required adaptation action.

(4) The route planning algorithm two is used when both the driver route preference and the traffic information are available. To represent this case, we define the adaptation rule four (R4) which have the availability of this context information as the condition to use the route planning algorithm two.

![Figure 14: Context-aware adaptive behaviour of the vehicle routing planning system](image)

(5) In Figure 14, we define R5 as a component that evaluates to true when the vehicle speed is greater than 70 km/h (i.e. the rule enabling condition), and has the adding (i.e. enabling) of the voice instruction component as the required adaptation action to reduce the driver distraction.

(6) When the driver is driving in low speed, the voice instruction may be annoying, and then it should be removed. In Figure 14, R6 evaluates to true when the vehicle speed is lower than 80 km/h, and has the removal of the voice instruction component as the required adaptation action.

### 4.2 Validating the Context-aware Adaptive Behaviour Visually

The system adaptive behaviour can be visually validated by choosing “Run the System Adaptive Behaviour Test” from the tool’s menu in our CAST tool. To enable this feature, we generate the system implementations and a code that makes an instance of these implementations and a GUI that is linked with this instance. This GUI visualizes the context providers, the context model, and the functional system. Using this GUI, the software engineer can change the context situation by providing specific context values in the displayed textboxes. Then, by pressing the “Adapt to the Context Information changes” button, the system implementation instance is adapted to the context changes and its state is displayed into the GUI.

Figure 15 shows an example, where the software engineer changes the driver route preference availability value and the route planning one state to be “active”. This context situation activates the adaptation rules one and three: (a) the context model is changed by removing the traffic information context entity and (b) the functional system is adapted by selecting the route planning algorithm one. By repeating this process, (a) missing adaptation rules can be detected, if providing a context situation has no reaction from the system; (b) incorrect adaptation rules can be detected, if context changes lead to unexpected system reactions.
4.3 Verifying the Adaptive Behavior Using Romeo Tool

For enabling the adaptive behaviour verification using the Romeo tool as discussed in section 3, we linked our tool with the Romeo tool. This link is through generating (a) the Petri Net model as an XML file and (b) the properties that need to be checked as TCTL file (i.e. the input files format of the Romeo tool). Then, we use the model checker implemented inside the Romeo tool for checking the adaptive behaviour, and then we get the verification results and display them in our tool in a user friendly manner. In addition, we enable the specification of the Petri Net initial marking using our tool through a GUI that visualized the context providers as shown in Figure 16-A. The specified context values are used for evaluating the adaptation rules condition, and then the condition that is evaluated to true its net input place is activated (i.e. have one token). When the software engineer press the verification button, the Petri Net model is generated, the model checker is called, and then the verification result is display as in Figure 16-B.

For example, when the vehicle speed is equal to 75 (Figure 16-A), the adaptation rules five and six is evaluated to true in the same time. The adaptation actions in this case are adding and removing the voice instruction component, and then a conflict is detected between the rules evaluated to true (i.e. R5 and R6) as shown in Figure 16-B.
By repeating the above process several times, we have detected the following errors in the specified adaptive behaviour:

**Error type 1**: The adaptation rules five and six actions can be triggered simultaneously when the vehicle speed value is between 70 and 80 km/h. Therefore, a conflicting action can happen (i.e. add then remove the voice instruction component).

**Error type 2**: The driver route preferences and the traffic information context entities can be active in the same. As consequence, the adaptation rules three and four are triggered together which means there are two replacements for the route planning algorithm in the same context situation.

**Error type 3**: The designed adaptive behaviour is free from redundancy error, where there is not duplication in the adaptation rules or there is an adaptation rule that is a part of another.

**Error type 4**: Rule one is to change the context model in response to the functional system change (i.e. remove the traffic information when the route planning one is active). In addition, when the traffic information is not available (i.e. the context change), the route planning one is selected (i.e. functional system changes). These two rules have cycles, where the activation of one rule makes the other active and vice versa. This leads to an infinite loop between them.

**Error type 5**: when there is not any context information available, the system should use the default route planning. However, the specified adaptive behaviour does not have this rule (i.e. missing adaptive behaviour). In addition, there are other missing adaptation behaviours, where we only show a simplified example to highlight the possible error.

## 5 Related Work and Discussions

In this report, we have proposed an approach for specifying and verifying the context-aware adaptive behaviour of a software system. In the following, we compare existing approaches and our approach with regard to the specification and verification of the system adaptive behaviour.

**The system adaptive behaviour specification**: There are three different approaches for specifying the system adaptive behaviour [13]. Firstly, the rule-based mechanism defines the system adaptive behaviour as a set of condition-action rules [8, 19-22]. These rules are used to define the required adaptation actions (i.e. the rule action) in response to the context changes (i.e. the rule condition). The condition-action rules (a) are easy to write (i.e. expressive); (b) do not need to define the possible system states (i.e. the possible system’s configurations and behaviours) beforehand such as the goal-based mechanism; (c) give a fast system reaction to context changes, where the needed system reactions are already defined. However, defining the specific system reactions to the context changes during the design time may be difficult in large scale systems with large number of adaptation behaviours. In addition, the adaptation rules are subject to errors such as inconsistency (e.g. applying two contradicting rules leads to inconsistent system state), incompleteness (i.e. missing adaptive behaviours), etc. Existing approaches do not provide a way tackle the above two issues [8, 19-22].

Secondly, the goal-based mechanism specifies the possible system’s configurations/behaviours as states. These states are used to build a state-based model for the system’s adaptive behaviour (e.g. Petri Nets [6], or Labelled Transition Systems [23]), where the transitions between these states are enabled by the context changes. In this approach, the specific system adaptation actions are specified at runtime by computing the difference between the system current state (i.e. configuration or behaviour) and the desired state. In addition, having a state-based model for the system adaptive behaviour enables its verification, and missing adaptation behaviours can be detected by looking for the missing transitions between the system states. However, when the number of the context variables (i.e. the context changes that the system adaptive behaviour model is based on) becomes large, the state explosion problem happened. Even if the model does not have the state explosion problem, the enumeration of all the possible system state is difficult and may be impossible. In
addition, comparing to writing the condition-action rules, the building of state based models is difficult. Furthermore, computing the required adaptation actions at runtime causes an overhead to the system, which affects the system performance, in particular systems that runs on low power devices.

Thirdly, the utility-based mechanism captures the system adaptive behaviour as a set of utility functions. These utility functions are used to evaluate the system variants in response to the context changes. Then, the variant that has the best utility is chosen as the system next state [11-12, 24]. Similar to the goal-based approach, the specific system adaptation actions are computed at runtime by computing the difference between the system current state and the desired state. In the case of the context change, the goal-based technique classify the possible next states to desired or not desired state (i.e. 0 or 1 classification), but the utility-based approach quantify each possible next state with a number between 0 and 1 based on the this next state suitability to cope with the context changes (i.e. generalization of the goal-based approach). However, when there are a large number of context variables that are used to define the utility functions, the design of the utility functions is complex. In addition, this approach has problems similar to the goal-based approach such as: (a) the need to enumerate all the possible system states at design time; (b) the runtime overhead where the utility functions are computed at the runtime.

Several approaches have been proposed for specifying the system adaptive behaviour, but these techniques still have some limitations as discussed above. We introduced an approach that have the expressiveness of rule-based mechanism (i.e. the easiness in writing condition-action rules), and the formality of the goal-based mechanism (i.e. the generated Petri Net model) to enable the adaptive behaviour verification. Therefore, it removes the limitation of existing approaches. In addition, for solving the state explosion problem, we classified the system possible adaptation to dependent and independent variations. This classification reduces the system state space and the transition between them as described in section three. Furthermore, existing approach capture the context model implicitly with the system adaptive behaviour model, and then the system model complexity is increased. In our approach we separate the system’s context model from its adaptive behaviour model and capture their relationship explicitly. As such, our approach captures the system adaptive behaviour easily while reducing the system modelling complexity.

**The system adaptive behaviour verification:** The goal-based approaches proposed for specifying the system adaptive behaviour enable the system adaptive behaviour verification, where they have a state-based model that can be verified. However, these approaches do not take into account the detection of errors that can happen during the adaptive behaviour specification [6, 23]. In addition, some of the adaptive systems frameworks support the functional system verification with regard to properties it should preserves and/or achieve, but they do not pay much attention to the system adaptive behaviour verification [6, 8, 10, 25].

Similar to our work, an approach has been proposed for verifying the context-aware adaptive behaviour of the mobile applications [7]. In their approach, they are concerned with the system parameter adaptation and not the system’s structure and behaviour changes, and then they do not consider the inconsistency type one error identified above. In addition, they assume the context model is fixed, but the context model can be changed during the runtime as shown on our case study. The changeability of the context model enables the cycles to happen in the system adaptive behaviour (see section three), which is not considered in their approach and need to be detected. Another assumption of their approach is the provided adaptive behaviour model is complete, and then they do not consider the completeness check. Finally, they consider the system state as the system whole configuration and/or behaviour, and then they face the state explosion problem as discussed above.
6 Conclusions and Future Work

In this report, we have proposed an approach to specifying and verifying the context-aware adaptive behavior of a software system. We have considered the context model and the system adaptive behavior model separately, and then their relationship is captured explicitly. To enable the system adaptive behavior model specification, we have introduced a component model that explicitly supports the definition of the system’s context and management (i.e. the adaptation rules conditions and actions). In addition for verifying the system adaptive behavior, we have identified a set of errors that can happen in specifying the system adaptive behavior and we transformed the specified model to Petri Net. Then, we used Romeo tool to perform the verification with regard to the errors identified. Furthermore, we have extended our prototype tool for automating the process of specifying and verifying the system adaptive behavior. We have also demonstrated our approach through specifying and verifying the context-aware adaptive behavior of the vehicle route planning system.

Compared to existing approaches, our approach has the following key contributions. First, our approach represents the relationships between the context changes and the system variations explicitly, so that the system adaptive behaviour is easily captured with less system modelling complexity. Second, we classify the possible system variations/adaptation into dependent and independent variations for reducing the possible system states and the transition between them (i.e. making the state explosion problem not easily reached). Finally, the designed system adaptive behaviour model is transform to Petri Nets so that it can be verified for detecting the adaptation behaviours errors such as inconsistency, redundancy, circularity, and incompleteness.

There are several future directions for this research. Firstly, in this paper we consider the verification of the system adaptive behaviour during the design time. We will extend our approach to (a) make the system able to add a new adaptive behaviour at runtime to cope with the unanticipated context changes and (b) enable the runtime verification of the system adaptive behaviour when a new adaptive behaviour is added. Secondly, not only the system adaptive behaviour needs to be verified but also the functional system itself, and then we will investigate the design time and runtime verification of the functional system. Finally, we have identified a set of errors (see section three) that can happen in specifying the system adaptive behaviour model. A more investigation will be performed to identify other possible errors if any.

References