Author:	Hussein, M., Han, J., Colman, A. & Yu, J.
Title:	An approach to specifying and validating context-aware adaptive behaviours of software systems
Conference name:	9th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (IEEE EASe 2012)
Conference location:	Novi Sad, Serbia
Conference dates:	11-13 April 2012
Place published:	United States
Publisher:	IEEE
Year:	2012
Pages:	1-10
URL:	http://ieeexplore.ieee.org/
Copyright:	Copyright © 2012.

An Approach to Specifying and Validating Context-aware Adaptive Behaviours of Software Systems

Mahmoud Hussein, Jun Han, Alan Colman, and Jian Yu
Faculty of Information and Communication Technologies, Swinburne University of Technology
PO Box 218, Hawthorn, 3122, Melbourne, Victoria, Australia
{mhussein, jhan, acolman, jianyu}@swin.edu.au

Abstract— **Context-aware adaptive software systems need to have models for their adaptive behaviour. These models specify systems' reactions to changes in their environments. In large scale software systems with high variability, an explosion in the number of the system's sates (i.e. the system's configurations or behaviours) and the transitions between them (i.e. the system adaptive behaviour) is introduced. As such, *specifying* the system adaptive behaviour and *assuring its correctness* are major challenges. In this paper, we introduce a novel approach to *specifying and validating* 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 can be easily captured. We also classify the possible system variations into dependent and independent variations to reduce the possible system states and the transition between them. To assure the adaptive behaviour correctness, the system adaptive behaviour model is transformed to a Petri Net model so that it can be validated to detect adaptation behaviour errors such as *inconsistency, redundancy, circularity, and incompleteness*. In addition, we demonstrate our approach though *specifying and validating* the context-aware adaptive behaviour of a route planning software system.**

Keywords - Context-awareness, self-adaptivity, adaptive behaviour modelling , validation, formal specification.

I. 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, validate, and realize* such systems that evolve at runtime [1-5].

Context-aware adaptive software systems need to have a mechanism that decides the system reactions in response to context changes (i.e. the system adaptive behavior). This mechanism needs 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 more complex the system adaptive behavior will be. In addition, multiple adaptation actions can be triggered concurrently where the environment variables are shared between the conditions that trigger the system adaptations. Therefore, the specifying and validating the system adaptive behavior are challenging tasks. Existing mechanisms to decide on the required adaptation actions in response to context changes can be 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 are used to infer the state the system should be in to cope with the environment changes [6, 10]; (3) *utility-based mechanism*, where it generalizes the goal-based mechanism by quantifying each possible next state the system can be in with a value between zero and one instead of classifying the states as desired next state or not (i.e. 0 or 1classification) [11-12]. On the one hand, rule-based approaches are expressive and easy to write, but are prone to error (e.g. rules redundancy, conflict, etc.) and need to be validated. On the other hand, the goal-based and the utility-based mechanisms (i.e. state-based approaches) are difficult to build and face the state explosion problem in large scale systems, but they can be validated for correctness [13]. As such, there is a need for an approach that combines the expressiveness and easiness of the rule-based mechanism and the formalism of the state-based mechanism to enable the adaptive behavior validation.

In this paper, we introduce a novel approach to *specifying* and *validating* the adaptive behavior of a software system. It has the expressiveness and easiness of the rule-based mechanism and the formality of the state-based approaches. In addition, we provide a graphical representation of the system adaptive behavior model that captures the relationship between the context changes and the system adaptation actions explicitly. Compared to existing work, our approach has the following novel features. Firstly, our approach represents the relationships between the context changes and the system variations (i.e. the system configurations and/or behaviours) explicitly, so that the system adaptive behaviour can be easily captured. Secondly, we classify the possible system variations into dependent and independent variations to reduce the possible system states and the transition between them, so that the state explosion problem cannot happen easily. From the system reconfiguration point of view, independent variations mean that the change of a system component is not dependent on the other components while dependent variations means the change of a component is depend on 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 transformed to a Petri Net model so that it can be validated to detect the adaptive behaviour errors

such as (a) *inconsistency*, where 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 response to context changes; (c) *circularity*, where the chain of fired adaptation actions keep repeating (i.e. an endless loop in the decision making mechanism); (d) *incompleteness*, where there is a context situation that has no reaction from the system (i.e. *missing* adaptive behaviour).

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

II. Resarch Motivation 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 road side 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 phone to get his preferences. The traffic information may or may not be available depending on the communication technologies installed and the availability of the road side 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 at 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 in response to the context changes. For example, adding the voice instruction component when the vehicle is moving at 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. 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 road side units, etc.), which affects the system operation and need to be taken into account. When there are a large number of environment variables that needs to be taken into account, the system adaptive behavior becomes complex and its design becomes a difficult and error-prone task. Therefore, there is a need for an approach that can be used to (a) easily *capture* the large number of environment variables and the system reactions to change into them and (b) *validate* the adaptive behavior to avoid the adaptation errors that can lead to an undesired system state while it is in operation.

III. The Approach

In this section, we first describe our approach to specifying the system adaptive behaviour. Then, we describe how we transform the modelled adaptive behaviour to a Petri Net [14] model to enable its validation.

A. Specifying the System Adaptive Behavior

To model the system adaptive behaviour, we introduced a component model (shown in Figure 1) [15-16]. It has three types of ports (i.e. *functionality*, *context*, and *adaptation action*) and an *enabling condition*. 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.

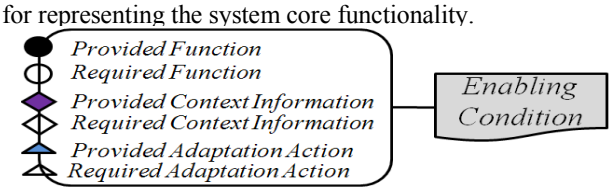

Figure 1. Our context-aware adaptive systems component model

In Figure 2, we show an example of the system adaptive behavior model using our component model. First, the whole adaptive behavior model is represented as a composite component (i.e. the change management), which has sub-components to represent the system reactions to the context changes. Each sub-component (e.g. *R2*) has the rule enabling condition(s) (e.g. *Attribute2 > 40*), and the rule actions as 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:

R1: *IF* the component one state is active, *THEN* the system removes the context entity two.

R2: *IF* the context attribute two value is greater than 40, *THEN* the system adds component two.

R3: *IF* the context entity two is inactive, *THEN* the system removes the adaptation rule two.

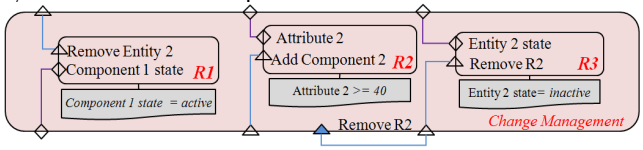

Figure 2. Specifying the system context-aware adaptive behaviour

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. 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 [13].

B. Validating the System Adaptive Behaviour

To validate the system adaptive behaviour using the existing model checkers (e.g. Romeo tool [17]), we need to (a) transform the specified adaptation rules to a state-based model (e.g. Petri Net [14]) and (b) specify the properties that need to be checked against the adaptive behaviour model (i.e. the errors that need to be detected).

Building the System Adaptive Behaviour State-based Model: The first step to building 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. From 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 consists of ten components where (1) two components can be added and removed when required; (2) two components have three variants and one of these variants is selected based on the system requirements; (3) six components are fixed which represent the system basic functionalities. The number of possible variations of this system will be 36 states (i.e. 2*2*3*3). Furthermore, if we consider a component from the six fixed components has three variants, then the total number of system states is increased to be 108 states (i.e. 3*36). Therefore, the number of the system states grows exponentially with the system possible adaptations, so 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, to take into account the components dependencies, we classified 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 the others. For example, the selection of a component realization is dependent on the component availability (i.e. is the component enabled or not). Then, for each dependent variation group, a state model is constructed. Finally, state models are constructed for each independent system component variations (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 realizations).

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 two active components, 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 for the change of component one realization, where there are four states when the component one realization two is selected and the other four states when the realization three is selected for component one.

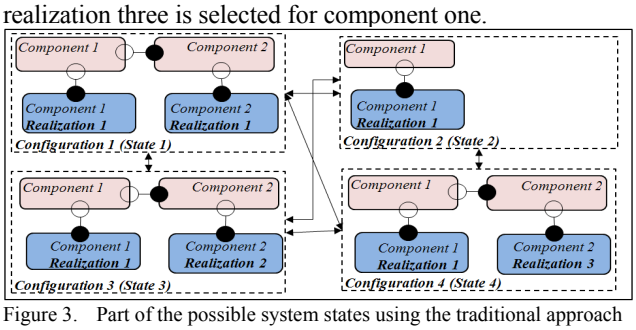

Figure 3. Part of the possible system states using the traditional approach

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

[1] In the following, our discussion is more concerned with the system reconfiguration as adaptation actions. The system behavior and parameters changes can be treated in the same manner.

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 a combination of components one and two states. 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).*

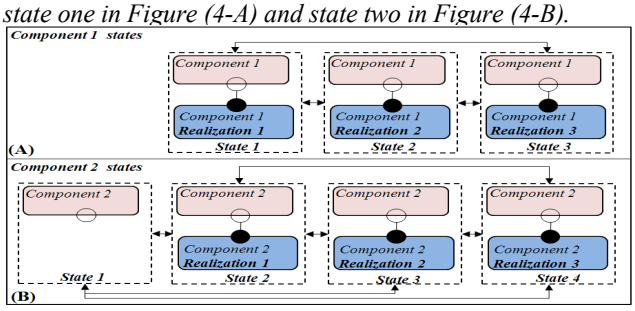

Figure 4. Possible system states using our approach

In addition to the reduction of the states from 12 to 7, the transition between these states (the system adaptive behaviour) is also reduced. There are 66 transitions with the model of 12 states using the traditional approach while we have 6 transitions within component one states model and 10 within the component two states model (i.e. a total of 16 transitions). In the following, we use the above approach to generate the Petri Net model from the adaptation rules and use it for detecting the adaptation behaviour errors.

The System Adaptive Behaviour Errors: In large scale software systems 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 by Romeo tool [17].

Adaptation Behaviors Inconsistency: The inconsistency means that the adaptation actions that need to be applied into the system contradict each other. The possible system adaptation actions are to add, remove, and replace 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 (*Type1 error*). Second, the required adaptation is to change (i.e. replace) the system element twice (*Type2 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 they can be detected.

1- Adaptation rules that have Type1 error:

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

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

In Figure 5, we transform the above rules to a Petri Net model (shown in Figure 5-A), where the rules conditions (e.g. VehicleSpeed_GreaterThan_40) are represented as *input places*, and the rules adaptation actions as *output places* (e.g. Add_VoiceInstructions). Each rule is captured using the Petri Net *transition* (e.g. R1) that links 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 the description 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 as a single place such as the traditional approach [6, 10].

To validate the inconsistency between these two rules using the generated Petri Net, we used Romeo tool [17]. On the one hand, 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 the rules one and two are activated in the same time (i.e. the final marking of the Petri Net as shown in Figure 5-B). On the other hand, to validate the rules formally, we represented the conflict Type1 error property using Timed Computation Tree Logic (*TCTL*) written in a format acceptable to Romeo tool [18] 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_ VoiceInstructions) and 6 (i.e. Remove_VoiceInstructions) 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 their adaptation actions contradict each other).

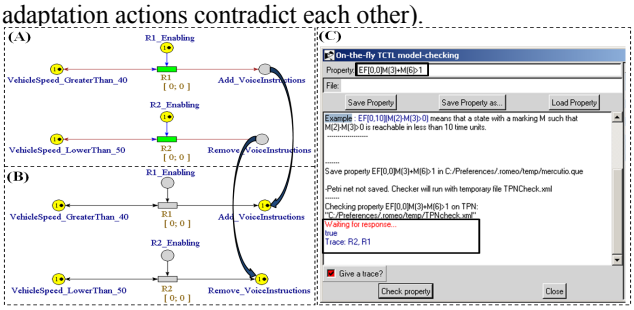

Figure 5. Petri Net for representing adaptation rules inconsistency

2- Adaptation rules that have Type2 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.

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 become inconsistent. To validate 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. the 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 fired together as shown in Figure 6-B and they are inconsistent adaptations.

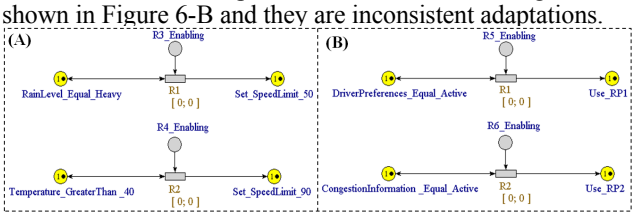

Figure 6. Petri Nets for representing adaptation rules three through to six

Adaptation Behaviours Redundancy: The redundancy appears when a rule is repeated, 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) (*i.e. Type3 error*). This error is detected by looking for an adaptation action that is repeated twice in the required adaptation actions.

Adaptation rules that have Type3 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 when the rules 1 and 2 are evaluated to true. The result is an adaptation script that has Add_VoiceInstructions action appeared twice. This error can be detected by checking the Petri Net using the *EF[0,0]M(3)>1* property. This means that "*is output place three activated twice (i.e. Add_ VoiceInstructions)*". The result shows that the evaluation of the adaptation rules one and two satisfy this property (i.e. the right part of Figure 7).

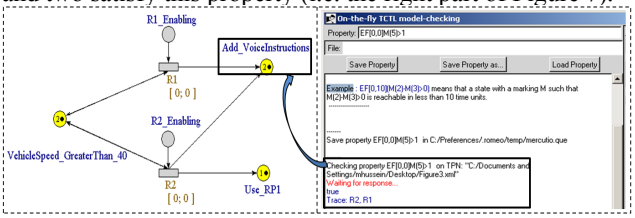

Figure 7. Adaptation rules that have the redundancy problem

Adaptation Behaviours Cycles: In context-aware systems, the context model changes initiate a system adaptation (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 adaptation (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). As such, the adaptation rules for changing the functional system in response to context model changes and vice versa should be written carefully to avoid the cycles. A cycle happens when the adaptation rules evaluation leads to adaptation actions that make the same chain of rules firing to be performed again (i.e. *Type4 error*).

Adaptation rules that have Type4 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* context entity.

Figure 8-A shows the Petri-Net where the rule one condition is active. Then, 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 keeps 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 one and two are fired continuously.

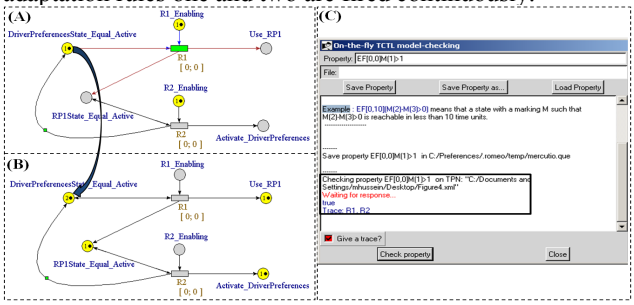

Figure 8. Petri Net for representing adaptation rules that have cycles

Adaptation Behaviours Incompleteness: In large scale systems, there are a large number of adaptation behaviours. As a consequence, there is a possibility of missing adaptation behaviours (i.e. *Type5 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 Type5 error:

R1: If the *driver preference is active and the congestion information is not active*, then the system uses the *route planning one*.

R2: If the *congestion information is active and the driver preference is not active*, then the system uses the *route planning two*.

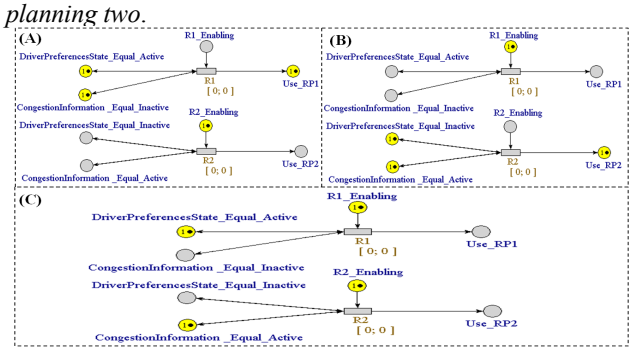

Figure 9. Adaptation rules that have the incompleteness problem

In Figure 9, we show the Petri Net that represents rules one and two and we show three different markings for the net. First, the condition of rule one is true (Figure 9-A). Second, the condition of rule two is true (Figure 9-B). Third a part of the conditions of rules one and two is true (Figure 9-C). We defined the completeness 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 5-A and 5-B satisfies this property where rule one or two is fired. However, Figure 5-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*).

[Missing Rule] R3: If the *congestion information is active and the driver preference is active*, then the system uses the *route planning three*.

By repeating the above process (i.e. putting an initial marking for the net and verifying the completeness property), the missing system rules can be detected (e.g. an adaptation rule that specifies the system reaction when both the driver preference and the congestion information is not active).

C. *Specifying and Validating the System Adaptive Behaviour using our Tool*

To support the specification and validation of the context-aware adaptive behaviour of a software system using our approach, we extended our **c**ontext-aware **a**daptive **s**ystems developmen**t** tool (*CAST*) [19]. This extension is to enable the software engineer to model and validate the system adaptive behaviour.

Specifying the system adaptive behaviour: The software engineer uses our component model to specify the system adaptive behaviour model. Then he feeds this model to our tool. Using our tool GUI, an adaptation rule can be specified as shown in Figure 10-A. This rule has a condition "*Vehicle_Speed > 40*" and adaptation action "*Add_VoiceInst*". This rule XML description is shown in Figure 10-B (details about our component model XML representation can be found in [15]).

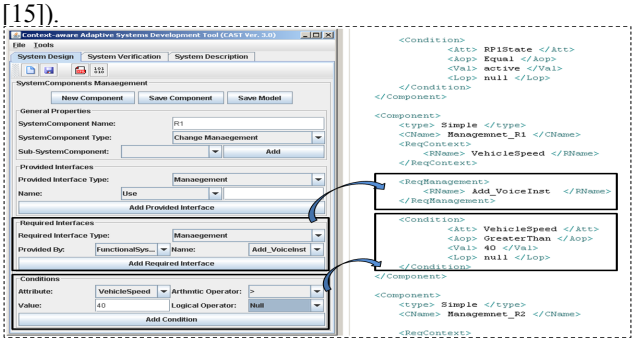

Figure 10. Specifying an adaptation rule using our tool

Validating the adaptive behaviour: Using our tool the software engineer can visually test the adaptive behaviour of the system (will be shown in the next section) and generate a Petri Net model that corresponds to the rule-based model as an XML file that is understandable by the Romeo tool. An example of a generated XML file that corresponds to the rule described in Figure 10 is shown in Figure 11. Figure 11-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 links these places together is shown in Figure 11-B. In Figure 11-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 be used to visually validate the net behaviour by playing the tokens games or using the TCTL model checker to validate this model formally according to the properties specified in the previous sub-section.

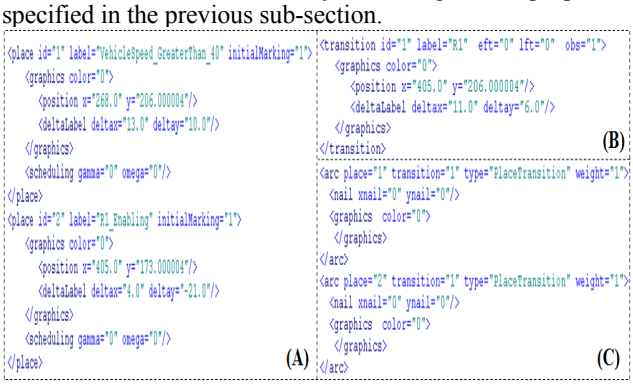

Figure 11. The Petri Net model description as an XML

IV. CASE STUDY

In this section, we specify and validate the context aware adaptive behaviour of the vehicle route planning system described in Section 2. We present the adaptive behaviour model in sub-section A, the visual validation of this model is discussed in sub-section B, and then its formal validation by Romeo tool is described in sub-section C.

A. *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. As such, in Figure 12, we show the system model that includes the context model, the functional system model, and the adaptive behaviour model using our component model. 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 12. 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 12. 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 planner 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.

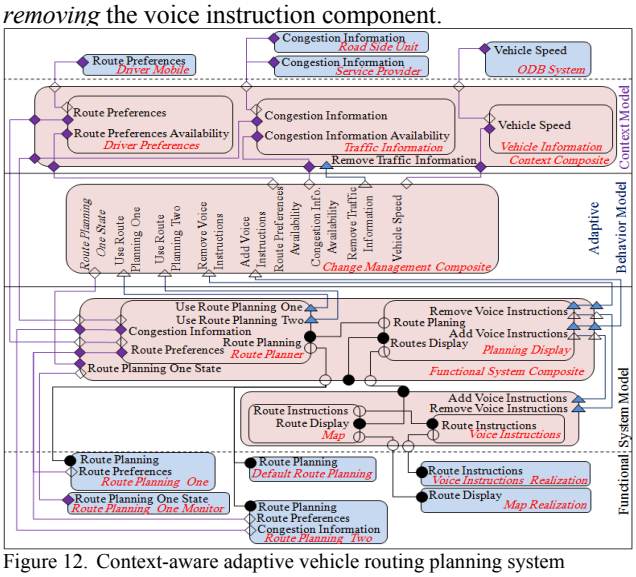

Figure 12. Context-aware adaptive vehicle routing planning system

The first component is realized in Figure 12 using 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.

The Adaptive Behaviour Model: The change management composite component (see Figure 13) 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 contain 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 13 has the *enabling condition* "is the driver uses route planning one (i.e. *active*)?" and the *required adaptation action* "remove traffic information" context entity.

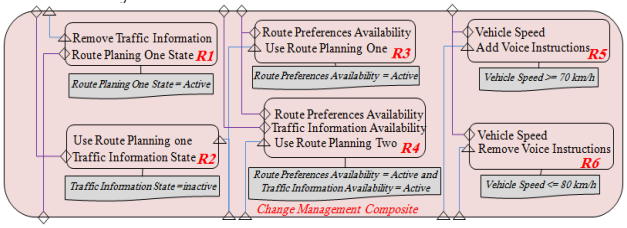

Figure 13. Context-aware adaptive behaviour of the vehicle routing planning system

(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 13) 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.

(5) In Figure 13, 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 addition (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 13, 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.*

B. Validating the 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.

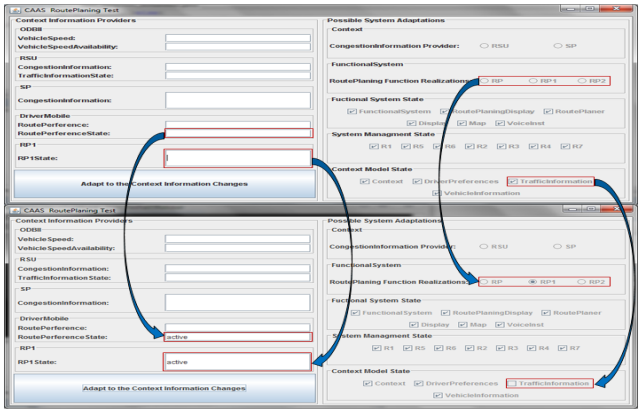

Figure 14. Testing the system adaptive behaviour visually

Figure 14 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 a context situation has no reaction from the system; (b) *incorrect* adaptation rules can be detected, if context changes lead to unexpected system reactions.

C. *Validating the Adaptive Behavior Formally*

For enabling the adaptive behaviour validation using the Romeo tool as discussed in section 3, we linked our tool with the Romeo tool. This link is made 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 15-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 validation button, the Petri Net model is generated, the model checker is called, and then the validation result is display as in Figure 15-B.

For example, when the vehicle speed is equal to 75 (Figure 15-A), the adaptation rules five and six are 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 15-B.

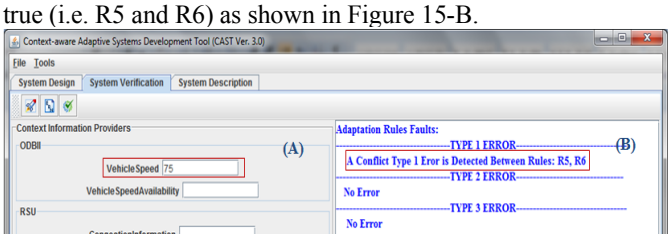

Figure 15. Validating the adaptive behaviour using Romeo tool

By repeating the above process several times, we have detected the following errors in the specified adaptive behaviour:

Type1 error: 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 and remove voice instruction component).

Type2 error: The driver route preferences and the traffic information context entities can be active at the same time. As a 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.

Type3 error: The designed adaptive behaviour is free from redundancy error, where there is no duplication in the adaptation rules.

Type4 error: 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. This leads to an infinite loop between them.

Type5 error: When all context information is not available, the system should use the default route planner. 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.

V. RELATED WORKS

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

The system adaptive behaviour specification: There are three different approaches to specifying the system adaptive behaviour [13]. Firstly, the *rule-based mechanism* defines the system adaptive behaviour as a set of condition-action rules [8, 20-23]. 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 state-based mechanisms; (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 a 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 to tackle the above two issues [8, 20-23].

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 [24]), 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 validation, 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 occurs. Even if the model does not have the state explosion problem, the enumeration of all the possible system states is difficult and may be impossible. In addition, compared 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 run 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, 25]. 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 response to context change, the goal-based technique classifies 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 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 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 has the expressiveness of a rule-based mechanism (i.e. the *easiness in writing condition-action rules*), and the formality of the state-based mechanisms (i.e. the *generated Petri Net model*) to enable the adaptive behaviour validation. Therefore, it removes the limitation of existing approaches. In addition, to solve the state explosion problem, we classified the system possible variations to dependent and independent variations. This classification reduces the system state space and the transition between themselves as described in section three. Furthermore, the existing approaches capture the context model implicitly with the system adaptive behaviour model which increases the system model complexity. 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 validation: The goal-based approaches for specifying the system adaptive behaviour enable the system adaptive behaviour validation, where they have a state-based model that can be validated. However, these approaches do not take into account the detection of errors that can happen during the adaptive behaviour specification and need to be detected [6, 24, 26]. In addition, some of the adaptive systems frameworks support the functional system validation with regard to properties it should preserve and/or achieve, but they do not pay much attention to the system adaptive behaviour validation [6, 8, 10].

Similar to our work, an approach has been proposed to validate the context-aware adaptive behaviour of the mobile applications [7]. In this approach, they are concerned with the system parameter adaptation and not the system's structure, 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 in our case study. The changeability of the context model enables the cycles to happen in the system adaptive behaviour (see section three) that is not considered in their approach and need to be detected. Another assumption of their approach is that 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.

I. CONLUSIONS AND FUTURE WORKS

In this paper, we have proposed an approach to *specifying* and *validating* the context-aware adaptive behavior of a software system. We have considered the context model and the system adaptive behavior model separately, so that their relationship can be easily captured. 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 actions (i.e. the adaptation rules conditions and actions). In addition, to validate the system adaptive behavior, we identified a set of errors that can happen when

specifying the system adaptive behavior and we transformed the specified adaptive behavior model to Petri Net. Then, we used Romeo tool to perform the validation with regard to the errors identified. Furthermore, we have extended our CAST tool for automating the process of specifying and validating the system adaptive behavior. We also demonstrated our approach through specifying and validating 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 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 transformed to Petri Nets so that it is validated 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 have considered the validation 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 validation of the system adaptive behaviour when a new adaptive behaviour is added. Secondly, not only the system adaptive behaviour needs to be validated but also the functional system itself, and then we will investigate the design time and runtime validations of the functional system. Finally, we have identified a set of errors that can occur when specifying the system adaptive behaviour model. A more investigation will be performed to identify other possible errors if any.

ACKNOWLEDGMENT

This research was partly supported by the Australia's Cooperative Research Centre for Advanced Automotive Technology (AutoCRC) *(www.autocrc.com)*.

REFERENCES

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.,* vol. 4, pp. 1-42, 2009.

[2] B. H. Cheng*, et al.*, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ed: Springer-Verlag, 2009, pp. 1-26.

[3] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," *Future of Software Engineering, 2007. FOSE'07,* pp. 259-268, 2007.

[4] S. Dobson*, et al.*, "Fulfilling the vision of autonomic computing," *Computer,* vol. 43, pp. 35-41, 2010.

[5] M. Huebscher and J. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Computing Surveys (CSUR),* vol. 40, p. 7, 2008.

[6] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006.

[7] M. Sama*, et al.*, "Model-based fault detection in context-aware adaptive applications," presented at the Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia, 2008.

[8] B. Morin*, et al.*, "Taming Dynamically Adaptive Systems using models and aspects," presented at the Proceedings of the 31st International Conference on Software Engineering, 2009.

[9] S. S. Andrade and R. J. de Araujo Macedo, "A non-intrusive component-based approach for deploying unanticipated self-management behaviour," in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, 2009, pp. 152-161.

[10] W. Heaven*, et al.*, "A Case Study in Goal-Driven Architectural Adaptation," in *Software Engineering for Self-Adaptive Systems*, ed: Springer-Verlag, 2009, pp. 109-127.

[11] R. Rouvoy*, et al.*, "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments," in *Software Engineering for Self-Adaptive Systems*. vol. 5525, B. Cheng*, et al.*, Eds., ed: Springer Berlin / Heidelberg, 2009, pp. 164-182.

[12] J. Floch*, et al.*, "Using Architecture Models for Runtime Adaptability," *IEEE Softw.,* vol. 23, pp. 62-70, 2006.

[13] Mahmoud Hussein, Jun Han, and A. Colman, "Context-Aware Adaptive Software Systems: A System-Context Relationships Oriented Survey," *Technical Report #C3-516_01, Swinburne University of Technology,* 2010.

[14] J. Peterson, "Petri nets," *ACM Computing Surveys (CSUR),* vol. 9, pp. 223-252, 1977.

[15] H. Mahmoud, H. Jun, and C. Alan, "An Approach to Model-Based Development of Context-Aware Adaptive Systems," in *InternationalConference on Computer Software and Applications*, Munich, Germany, 2011, pp. 205-214.

[16] H. Mahmoud*, et al.*, "An Architecture-based Approach to Developing Context-aware Adaptive Software Systems," presented at the 19th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS 2012), Novi Sad, Serbia, 2012.

[17] D. Lime*, et al.*, "Romeo: A parametric model-checker for petri nets with stopwatches," *Tools and Algorithms for the Construction and Analysis of Systems,* pp. 54-57, 2009.

[18] T. Hafer and W. Thomas, "Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree," *Automata, Languages and Programming,* pp. 269-279, 1987.

[19] *http://www.ict.swin.edu.au/personal/mhussein/CAST.htm,* 2010.

[20] Q. Z. Sheng and B. Benatallah, "ContextUML: A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services Development," presented at the Proceedings of the International Conference on Mobile Business, 2005.

[21] T. Gu, H. K. Pung, and D. Q. Zhang, "A service-oriented middleware for building context-aware services," *J. Netw. Comput. Appl.,* vol. 28, pp. 1-18, 2005.

[22] K. Henricksen and J. Indulska, "A Software Engineering Framework for Context-Aware Pervasive Computing," presented at the Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04), 2004.

[23] A. Mukhija and M. Glinz, "The casa approach to autonomic applications," in *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, Paris, France 2005.

[24] D. Sykes*, et al.*, "From goals to components: a combined approach to self-management," presented at the Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, Leipzig, Germany, 2008.

[25] D. Garlan*, et al.*, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer,* vol. 37, pp. 46-54, 2004.

[26] Y. Zhao*, et al.*, "Model Checking of Adaptive Programs with Mode-extended Linear Temporal Logic," in *2011 8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe)*, 2011, pp. 40-48.