Scenario-based Validation of Requirements for Context-aware Adaptive Services

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

Abstract— Context-awareness and adaptability are highly desirable features for services that are operating in dynamic environments. Recently, a number of approaches have been introduced to support the development of such services. But, validating the varying requirements of these services is still a major challenge. In this paper, we introduce a novel scenario-based approach to address this challenge. First, our approach captures a service’s requirements as two sets of scenarios: functional and adaptation. The functional scenarios represent the service’s core functionality, while the adaptation scenarios capture the service’s runtime adaptation in response to context changes. The service properties that need to hold at runtime are also represented graphically in a form similar to the scenarios. Second, a technique is introduced to enumerate and generate the specifications of a service’s variants from its scenarios. The generated variants are then validated against the service properties to ensure their validity. This technique also checks the consistency of the service’s adaptation requirements (scenarios). Case studies have shown that with our approach, a small number of service scenarios specified by the software engineer is able to cover a large number of service variants, which are generated and validated automatically.

Keywords- Adaptation Scenarios, Context-awareness, Service Adaptation, Variants Generation, Service Validation

I. INTRODUCTION

There is an increasing demand for services that are able to dynamically adapt their behaviours at runtime in response to changes in their environments. A challenge is how to validate such context-aware adaptive services [1-2].

Recently, a number of approaches have been introduced to support the validation of the varying requirements for context-aware adaptive services (e.g. [3-6]). However, two major challenges still remain. First, to validate a service’s varying requirements, the software engineer is responsible for identifying and specifying the service variants manually. For a complex service, this task is tedious and error prone, where the service’s high variability introduces an explosion in the number of its variant specifications. Second, existing approaches specify runtime adaptations to the functional scenarios in response to context changes. Thus, in this paper, we extend this technique to represent the service properties (that need to hold when the service adapts at runtime) in a form similar to the scenarios, so that the properties can be easily specified. To capture a service’s scenarios and its properties, we extend the UML sequence diagram with three types of participants (functional, contextual, and management) and some specific message types for the interactions between these participants. The functional participants provide the service functionality. The management participants decide and perform adaptation actions on the service’s functional scenarios. The contextual participants provide context information that is needed by the service functionality, or that triggers the service adaptation.

Second, to validate a service’s requirements (specified as two sets of functional and adaptation scenarios), we propose a technique that uses the service’s adaptation scenarios for enumerating and generating specifications of the service variants as determined by context changes. The consistency of the adaptation scenarios is also checked to ensure that the adaptation actions specified in these scenarios are free from errors such as redundancy, conflict, and incompleteness. The service variants and their properties are then transformed to Petri nets [10] and computational tree logic (CTL) formulas [7], before being fed into the Romeo model checker [11] for the conformance check.

A graphical tool has been developed to support the software engineer in validating the varying requirements of context-aware adaptive services. Case studies have also been performed to demonstrate the effectiveness of our approach in validating a large number of service variants.

The contribution of this paper is twofold: (1) a scenario-based technique to specify a service’s properties graphically, so that these properties can be easily and clearly captured; (2) a technique to generate a service’s variant specifications from its functional and adaptation scenarios, and to check the consistency of the service’s adaptation scenarios.

The remainder of the paper is organized as follows. Section 2 introduces a motivating scenario. In Section 3, we discuss a technique to specifying the requirements and the properties of context-aware adaptive services. Section 4 presents a technique to validating a service’s variants, and to checking the consistency of a service’s adaptation scenarios. The tool that has been developed to support our approach is presented in Section 5. The application of our approach to three case studies is discussed in Section 6. Section 7 analyses existing work with respect to our approach. Finally, we conclude the paper in Section 8.

1 The term variant is used throughout the paper to describe a service’s requirements specification in a specific context situation.
II. MOTIVATING SCENARIO

Let us consider a travel guide service that a tourist can use to plan a trip on her mobile. In the morning, based on her preferences (e.g. outdoor attraction) and the weather forecast for that day (e.g. sunny), the travel guide service suggests a number of attractions. She selects some of these attractions for visit. Then, the route planner of the travel guide service suggests to her a number of routes, which are calculated based on her location, her driving preferences, the selected attractions, and the traffic information. She selects a suitable route and starts to explore the attractions. Nearing the lunch time, the service suggests a number of nearby restaurants that match her food preferences.

To develop the travel guide service described above, two types of requirements need to be specified: functional and adaptation. The functional requirements are functions that need to be provided by the service (e.g. finding attractions). These functions also take context information into account to give better suggestions. For example, the attractions finder needs the weather information to suggest attractions that suit the weather conditions. The adaptation requirements specify the service adaptation in response to context changes. For example, the travel guide provider may want to provide the attractions finder for free, while the tourist should pay to use the other functions. Consequently, when the tourist wants to use and pay the route planner, the route planner needs to become available to the tourist while the travel guide service is in operation. In addition, to ensure that the travel guide service works properly, a number of properties need to be preserved while such adaptations occur. Example properties are: (1) the tourist must login to the service before using any of its functions; (2) the tourist cannot use the route planner when it is not in his selected features.

In general, not only the service’s functional requirements need to be specified, but also its adaptation requirements and properties that need to hold at runtime. In addition, variations in the service’s functional requirements triggered by context changes need to be validated for conformance to the service properties. In the following, we present the details of our approach for specifying and validating the requirements of context-aware adaptive services.

III. SPECIFYING THE SERVICE’S REQUIREMENTS AND PROPERTIES

To specify the requirements of a context-aware adaptive service, we introduced a scenario-based technique [9]. Using this technique, the service requirements are specified as two sets of scenarios: functional and adaptation. In this paper, we extend this technique to capture the service properties (that need to hold at runtime) in a form similar to the scenarios to facilitate the validation of the service’s varying requirements.

A. Background: Specifying the Service Requirements

To specify the requirements of a context-aware adaptive service as two sets of scenarios: functional and adaptation, we have adopted and extended the widely accepted UML sequence diagram [9]. Before introducing our extensions, we describe below the core elements of the UML sequence diagram.

(1) The UML Sequence Diagram: Following the UML sequence diagram, a scenario consists of lifelines, messages, and interaction fragments (see the top part of Figure 1). The lifelines represent the scenario participants. The messages capture the interactions between the participants. To capture the scenario flow, a set of interaction fragments are used such as an interaction use, an occurrence specification, and a combined fragment. First, the “interaction use” specifies that a scenario can refer to (or use) another scenario. Second, the “occurrence specification” concerns the intersection point between a participant and a message, defining what events are sent or received by a participant. Third, the “combined fragment” groups a number of interactions to define their relationships (e.g. loop, alternative, etc.). Each combined fragment consists of an operator and operands (see Figure 1). The operator defines the type of the combined fragment (e.g. alternative), while the operands specify interactions that are grouped by the combined fragment operator. In addition, each operand can have a constraint as a Boolean expression that specifies when this interaction can be executed.

![Figure 1. A meta-model for the extended UML sequence diagram](image)

(2) Specifying the Service’s Functional Requirements: To capture a service’s functionality while taking context information into account as a set of scenarios, we extended the UML sequence diagram by having two types of scenario participants (i.e. functional and contextual) that interact with each other through special types of messages as shown in the bottom part of Figure 1.

2.1 Functional Participants and their Messages: The functional participants are responsible for providing the service functionality. For example, the attractions finder participant (see Figure 2-B) is responsible for suggesting attractions to the user. The messages between the functional participants are either requesting or providing a functional operation, and therefore called “functional messages”.

---

2 Message types that can be exchanged between scenario participants depend on the types of the participants.
2.2 Contextual Participants and their Messages: Context information is needed by a context-aware service to better carry out its functionality [1]. Thus, we make the contextual participants explicit in the functional scenarios to provide the context information (e.g. weather information provider in Figure 2-B). The functional participants can be notified of context changes or they can request the context information on-the-fly. Thus, we have two types of contextual messages (i.e. get and notify context) that can be exchanged between functional and contextual participants. For example, the user can be notified of the weather information from its provider (i.e. CM1: “notify context” shown in Figure 2-B). To cope with runtime changes in this type of context information, an alternative fragment can be used where different interactions are executed based on the context status. For example, the interaction “FM1: FindIndoorAttractions” is executed when the weather is rainy, while “FM2: FindOutdoorAttractions” is executed otherwise as shown in Figure 2-B.

(3) Specifying the Service’s Adaptation Requirements: To represent a service’s adaptation requirements, we proposed the concept of adaptation scenarios. Such scenarios describe adaptations to the service’s functional scenarios in response to runtime context changes. To define this type of scenarios, we introduced management and functional service lifelines, management messages, and adaptation fragments as shown in the bottom part of Figure 1. The adaptation scenarios also have contextual participants.

3.1 Functional Service and Contextual Participants: In an adaptation scenario, the functional service participant represents the service’s functional requirements (scenarios) that need to be adapted in response to context changes (e.g. the travel guide service’s functionality “FL1” in Figure 3). The adaptation scenario also has contextual participants that provide context information that triggers the service adaptation. For example, changes in the features selected by a user cause the travel guide service’s adaptation. Thus, we have a contextual lifeline “CL1” to make this information available as shown in Figure 3.

3.2 Management Participant and its Messages: In an adaptation scenario, a management participant is responsible for deciding and performing the service adaptation to cope with context changes (e.g. the organizer participant “ML1” shown in Figure 3). The management participant uses a set of management messages to adapt the service functionality, which specify what adaptation actions that can be applied to the service’s functional scenarios. Example actions that can be applied to the scenario “FS2” (shown in Figure 2-B) are: remove the message “CM1”, and add the lifeline “CL1”. In general, we have actions to add, remove, or modify all the scenario elements (lifelines, messages, combined fragments, etc.) as shown in Figure 1.

3.3 Adaptation Fragment: To specify what to adapt in response to a context change, we introduced the adaptation operator as shown in Figure 1. This operator groups a set of management messages (i.e. adaptation actions). We also extended the interaction constraint with context condition (see Figure 1) to specify a contextual situation in which the service needs to adapt itself (e.g. the user wants to include the route planner function). We call the combined fragment that has the adaptation operator and the context condition as
“adaptation fragment”. An example adaptation scenario is “AS1” shown in Figure 3. It has three adaptation fragments that customize the service based on the features selected by the tourist. For example, the fragment “AD1” specifies the changes to the functional scenario “FS1” based on the user’s need to include or exclude the route planning function.

B. Specifying the Service Properties

At runtime, a set of service (temporal) properties need to be preserved to ensure that the service is working properly before and after its adaptation. To represent these properties, we use the property specification patterns [12]. The patterns provide a high-level easy-to-use notation for defining commonly occurring properties that would otherwise require formalisms such as computational tree logic (CTL). An example pattern is “Absence” that means an event does not occur during the service execution. Each pattern also has a scope to define when the pattern should hold. For example, the scope “Globally” specifies that a pattern must hold during the entire service execution. More details about these patterns can be found in [12].

Similar to capturing a service’s functional or adaptation requirements as a set of scenarios, we extend the sequence diagram to allow the specification of service properties in a form similar to the scenarios. We first introduce two new interaction operators: pattern and pattern scope, to represent the property patterns and their scopes (see Figure 1). Then, the property operator is introduced to define combined fragments consisting of a pattern fragment (defined using the pattern operator) and optionally a pattern scope fragment (defined using the pattern scope operator). When the pattern scope is not specified, the default globally scope is assigned to the property. The graphical representations for these operators and fragments are given at the right of Figure 4.

Figure 4. Example properties of the travel guide service

The service has multiple runtime variants and it switches between them in response to context changes. As such, the service has two types of properties: local and global [5]. The local properties need to hold only with specific variants, while the global properties need to hold in all service variants (i.e. independent of context situations). To specify the local properties, we incorporate context conditions into the property patterns to specify the context situations where these properties should hold (see Figure 1).

Figure 4 shows some example service properties. First, the property “PD1” is a global property and must hold in all service variants, i.e. the “logout” operation should be always preceded by the “login” operation. Second, the property “ET1” is a local property and should only hold when the attractions finder is selected, meaning that the interaction “ProvideAttractions” should exist (i.e. the pattern) after the interaction “FindIndoorAttractions” has been requested (i.e. the scope “AF1”).

IV. VALIDATING THE SERVICE REQUIREMENTS

While a context-aware adaptive service in operation, it switches from one variant to another in response to context changes. During this adaptation, a set of service properties need to be preserved. Therefore, the service variants need to be validated against the relevant service properties.

To validate a service’s variants, we generate the service variant specifications from its requirements (scenarios). The generated variants are then transformed to formal models to check their conformance to the relevant properties.

A. Generating the Service Variant Specifications

To generate the service variants, we compute adaptation actions (as scripts) that need to be applied into the service in response to context changes. These scripts are then checked to ensure their consistency and applied into the functional scenarios to generate the service variants.

1) Computing the Adaptation Scripts: To compute the adaptation scripts, we first enumerate the possible context situations in which the service may need to adapt itself. A contextual situation is defined as a set of conditions on the context variables, which correspond to the conditions of the adaptation fragments. We identify the conditions by parsing the adaptation scenarios. If an identified condition does not already exist in the accumulating list of conditions (i.e. a new condition), it is added to this list (see Lines 2-9 of the algorithm presented in Listing 1). The result of this step is a set of conditions that trigger the service runtime adaptation. For example, applying this part of the algorithm to the adaptation scenario “AS1” shown in Figure 3 generates the following list with three conditions:

“([RoutePlannerSelected==True],
(FindingAttractionSelected==True),
(RestaurantLocatorSelected==True))”.

Second, after collecting the conditions that can trigger the service adaptation, we generate all combinations of these conditions and remove the invalid ones. Each condition can be true or false. Thus, the number of combinations equals to \(2^n\) where \(n\) is the number of the identified conditions. To assign Boolean values to one condition combination, we use the binary representation of the combination index with \(n\) binary digits (see Lines 10-14 in Listing 1). For example, the second combination (i.e. the index is 1) of the conditions generated from the scenario “AS1” is “001” which means:
"RoutePlannerSelected==True" is false, "FindingAttractionSelected==True" is false, and "RestaurantLocatorSelected==True" is true.

Then, invalid combinations that have conflicting conditions assigned the same Boolean value (i.e. conditions that cannot be true or false at the same time) are identified and removed as shown in Line 15 of Listing 1.

Third, the valid combinations of adaptation conditions are used for generating the adaptation scripts (each of which corresponds to a service variant). For each combination, we traverse the adaptation fragments in the scenarios to identify executable ones (i.e. their conditions are evaluated to true). Then, the adaptation actions specified in these executable fragments are added to an adaptation script (see Lines 18-23 in Listing 1). An example script is "S2":

```
[[RemoveCombinedFragment(FS1, RF1),
  RemoveCombinedFragment(FS1, RF2), and
  AddCombinedFragment(FS1, RF3)]]
```

which is generated from the second condition combination.

Listing 1. An algorithm (pseudocode) to validate the service requirements

1. Validating the Service Requirements (FunctionalScenarios FS, AdaptationScenarios AS)

   // Compute the adaptation conditions
   2: List ConditionList = new list();
   3: FOR each Scenario S in AS
   4:    FOR each AdaptationCondition AC in S
   5:      IF AC does not exist in the ConditionList THEN
   6:        ConditionsList.add(AC);
   7:    END IF
   8:  END FOR
   9: END FOR

   // Generate the possible conditions' combinations
   10: List ConditionsCombinations = new List();
   11: ConditionsCombinations.setSize(power(2, sizeof(ConditionsList)));
   12: FOR each Combination C in ConditionsCombinations
   13:    C.assignBooleanValue(BooleanValueOf(C.index()));
   14: END FOR
   15: ConditionsCombinations.RemoveInvalidCombinations();
   16: FOR each Combination C in ConditionsCombinations
   17:    AddCombinedFragment(FS1, RF3); // having same value. E.g. If C1 is A>1 and C2 is A<1,
   18:      ConditionalExpression(C1==C2, C1==C2);
   19:    END IF
   20: End FOR

   // Generate and check adaptation scripts
   21: List AdaptationScripts = new List();
   22: FOR each Combination C in ConditionsCombinations
   23:    FOR each Action A in C
   24:      AddCombinedFragment(FS1, RF3); // The sequence references "RF1" (plan a route)
   25:      AddCombinedFragment(FS1, RF2); // and "RF2" (finding attractions) have been removed where
   26:      END FOR
   27: END FOR

(2) Checking the Adaptation Scripts: An adaptation script is consistent, if it is free from redundancy, conflict, and incompleteness in its adaptation actions. To ensure the consistency of a generated script (Line 24 in Listing 1), we first identify redundant actions in the script, where an action may be fired by two adaptation fragments at the same time.

Second, conflicting actions (i.e. adaptation actions that cannot be fired at the same time) are detected. For example, a generated script may have two actions to add and remove an element, or to modify an element twice in a functional scenario. Thus, we parse each generated script to detect such conflicts in its actions.

Third, the script actions are checked for missing actions (i.e. actions that should appear in the script but they do not). These actions are inferred from the dependencies between the actions. In general, a combined fragment depends on its messages and a message depends on the lifelines involved. Thus, to remove an element from a scenario, elements that depend on this element must be removed first. For example, to remove the last message in a combined fragment, the fragment should also be removed. Similarly, to add an element to a scenario, elements that depend on this element depends on should be added first (e.g. before adding a message, the lifelines that send and receive this message must be added). We use these action dependencies to detect missing actions, where dependent actions should coaxist with each other. We do so by parsing each adaptation script to identify actions whose dependent actions do not exist in the script. For example, a script may have an action to remove a lifeline, but it does not have actions to remove messages that the lifeline is involved in.

For each inconsistent script (i.e. one with redundant, missing or conflicting actions), the engineer is notified by the adaptation scenarios that lead to this inconsistent script, so that he can perform the required changes to the scenarios.

(3) Deriving the Service Variants: For each consistent adaptation script from the above step, its adaptation actions are first automatically ordered to take the dependencies between the actions into account. Then, a service variant is generated by applying this script to the service’s functional scenarios (see Lines 28-31 in Listing 1). An example generated variant is shown in Figure 5-A, which is the result of applying the script “S2” described above to the scenario “FS1”, where the sequence references “RF1” (plan a route) and “RF2” (finding attractions) have been removed where they are not needed by the user. Note that the action “add combined fragment RF3” does not have an effect on the scenario “FS1”, because this scenario contains all elements that the service needs to have at runtime (see Figure 2-A).

To identify properties that need to be checked against a generated variant (see Line 32 in Listing 1), we first parse the service properties. If a property does not have a context condition (i.e. global), then it should be satisfied by this variant (e.g. the property “PD1” shown in Figure 5-A). On the other hand, if a property needs to only hold in a specific context situation (i.e. local), then the context situation in which this variant needs to be active is compared with the property’s context condition. If the property condition is satisfied, then this property needs to hold in this service variant (e.g. the property “ABI” shown in Figure 5-A).

B. Validating the Service Variant Specifications

To validate the service variant specifications against the relevant service properties, we transform the variants and their properties to formal models. In this regard, we choose
to use the Romeo tool to perform the formal validation [11]. Therefore, we transform the variants to Petri nets and the properties to computational tree logic (CTL). We adopt the technique proposed by Bernardi et al. [13] to transform each scenario in a service variant to a Petri net. For example, the variant of the scenario “FS1” (see Figure 5-A) is transformed to a corresponding Petri net shown in Figure 5-B (more details about this transformation process can be found in [13]). We also use the mappings introduced by Dwyer et al. to transform the service properties to CTL formulas [12]. For example, the precedence property “PD1” between login and logout shown in Figure 4 is transformed to “\texttt{not E [not login U (logout and not login)]}”. Finally, the Petri nets and the CTL formulas are fed into the Romeo tool for validation (Line 33 of Listing 1). If the properties are not satisfied by the variants, the software engineer is alerted, and corrective actions need to be carried out to the scenarios.

![Figure 5. An example variant generated from the scenario “FS1”](image)

V. TOOL SUPPORT

To support the validation of the varying requirements for context-aware adaptive services, we have developed a tool (see Figure 6 for a screenshot). It helps the software engineer in performing a number of tasks. First, it assists the engineer in specifying a service’s requirements and its properties. Second, the tool automatically generates possible adaptation scripts from a set of adaptation scenarios, checks the scripts to identify inconsistencies, and notifies the software engineer by the scenarios that lead to these inconsistencies.

Third, to check a service’s variant specifications against their properties, the tool enables the generation of the service variants from its functional scenarios and the transformation of these variants to Petri nets and their properties to CTL formulas in a format that is acceptable by the Romeo tool [11]. Our tool also communicates with the Romeo model checker to perform the validation. Then, the tool displays the validation results to the engineer. Based on these results, the software engineer may change the service scenarios and properties as appropriate and repeat this task until all service variants are valid.

![Figure 6. Screenshot from the developed tool](image)

VI. CASE STUDIES

In this section, we discuss the application of our approach to three case studies, and analyze its effectiveness in assisting the software engineer in validating the varying requirements of context-aware adaptive services. The results of the three case studies show that a small set of scenarios specified by the software engineer are able to cover a large number of service variants which are generated and validated automatically.

A. The Travel Guide Service

We have used our scenario-based approach for specifying and validating the travel guide service described in Section 2.

(1) Specification of the Service Requirements: The requirements for the travel guide service are specified as 4 functional scenarios and 4 adaptation scenarios. First, the functional scenario for the main flow of the service is shown in Figure 2-A, while the other three scenarios capture the user interactions with the travel guide service to plan a route, find attractions (Figure 2-B), and locate a restaurant. Second, each functional scenario has a corresponding adaptation scenario. The four adaptation scenarios have 8 adaptation fragments that specify runtime changes to the functional scenarios in response to changes in the availability of the traffic and the weather information, and the user’s selected features. The service also has 5 global properties and 16 local properties that need to be preserved at runtime (two of these properties are shown in Figure 4).

(2) Validating the Requirements: Using the algorithm described in Section 4, 8 adaptation conditions are identified and then there are 256 (2^8) possible combinations of these conditions. The adaptation fragments are specified carefully. Thus, there are no combinations with conflicting conditions. All the 256 adaptation scripts generated are also consistent leading to 256 service variants. To show the ability of our algorithm in identifying inconsistencies and errors in the adaptation scenarios, we have added two wrong fragments. The first fragment’s condition is “RoutePlannerSelected == False” which conflicts with the condition of the fragment “AD1” shown in Figure 3, and the second fragment’s condition is same as the condition of the fragment “AD2” in Figure 3 but its action is “RemoveCombinedFragment (FS1, RF2)” that contradicts to the action of “AD2”. After adding these two fragments, the number of condition combinations becomes 1024 (2^10). Half of these combinations are invalid and removed, because the two conflicting conditions cannot have the same Boolean value at the service runtime. The remaining 512 combinations lead to 512 adaptation scripts.
However, there is a contradiction between the actions of two adaptation fragments, and then half of the generated scripts are removed where the two fragments cannot be executed at the same time, and the errors are reported. The algorithm is able to generate 256 variants (see the third column in Table 1). After being transformed to formal models, these variants and the service properties are fed into the Romeo tool for the formal validation.

### B. The Mine Pump Service

We have used our approach to validate a commonly used example in the literature “a mine pump service” [14]. This service monitors and controls the water level in a mine to prevent water overflow. It has three functional scenarios and two adaptation scenarios. The first functional scenario describes the service’s main flow. The other two scenarios describe the automatic operation of the mine pump service, and the user’s manual interactions with the service. The pump service has three operating modes based on the water level (low, medium, and high). Thus, an adaptation scenario is defined to switch between these three modes. Another adaptation concerns the alarm and command interfaces as optional features in the service. As such, a further adaptation scenario is defined to adapt the service based on the needed features. These scenarios have 6 adaptation conditions, and therefore 64 \((2^6)\) possible combinations of the conditions and 64 service variants are generated (see Table 1).

### C. The Electronic Exam Service

Our approach is also used for validating a larger service: an “electronic exam (e-exam) management service” based on the example in [15]. This service enables a lecturer to design a computer-based exam, the students to take the exam, and the lecturer (or an assessor) to score the exam. The e-exam service needs to be used by different universities, lecturers, and students. Thus, there is a need to develop an e-exam service that can be customized to suit different universities’ requirements, and is able to be adapted at runtime based on its users’ context (e.g. a student can only take an exam when he is at the exam location). Table 1 presents the number of service variant that are enumerated and generated from the functional and adaptation scenarios of the e-exam service.

### VII. RELATED WORK

In this section, we discuss approaches concerning the specification or validation of requirements for context-aware adaptive services, and compare them with our approach.

1. **Specification of the Service Requirements:** RELAX is an approach to capture an adaptive service’s requirements by relaxing the service requirements to take into account context changes [16]. They first capture the service requirements (goals) as SHALL statements in a textual representation. Then, they classify these requirements as relax-able (i.e. changeable at runtime to take into account context changes) or invariant (i.e. cannot be violated at runtime) requirements. Finally, they apply RELAX operators (e.g. as early as possible) to the relax-able requirements, so that they become flexible to cope with runtime context changes. Another approach is introduced by Salifu et al. to capture a service’s monitoring and switching requirements [17]. The monitoring requirements are context variables that trigger the service adaptations, while the switching requirements specify the service’s switching between its different variants to keep achieving the service goals in the face of context changes. To capture the context information and its effect on the service, they adopt problem frames approach.

   These approaches are based on specifying the service requirements as a set of goals. However, it is often the case that the stakeholders find it difficult to articulate their needs as a set of goals [18]. Thus, the requirement engineer resorts to capture information about tasks that the users perform or want to perform as a set of scenarios [19]. On this basis, we adopt a scenario-based approach in specifying the functional and adaptation requirements of a context-aware adaptive service. In addition, in the exiting approaches, the service’s contextual and adaptation requirements are intertwined with its functional requirements specification. As such, specifying the service requirements using these approaches is a complex task. In our approach, we represent the service functionality, context, and adaptation explicitly, so that these aspects and their relationships can be clearly captured.

2. **Specification of the Service Properties:** To specify a service’s properties, a number of formal notations have been introduced (e.g. [7-8]). But, the service properties that can be easily expressed in natural language are hard to be captured by these formalisms. In addition, the task of writing such formalisms is tedious and error prone. Autili et al. proposed an approach to capture a service’s properties by an extension to the UML sequence diagram [20]. But, in this approach, the properties are specified into the functional scenarios. In our approach, we specify these properties separately from the functional scenarios, so that the properties can be easily and clearly specified. In addition, we enable the specification of context situations in which properties should hold, which is not considered in existing approaches.

3. **Validating the Service Variants:** Fuxman et al. introduced an approach to validate a service’s requirements captured by the Tropos notation [3]. They proposed a language to formalize the requirements model which they call Formal Tropos. In addition, a software tool is developed to transform the requirements specification to a finite state machine that is acceptable by the NuSMV model checker. The NuSMV is then used to check the requirements model against its constraints. Lettieri et al. introduced an approach to enable the validation of a service’s requirements specified as a goal model [4]. They introduced a formal representation of the goal model. This formal model is then used to derive a labeled transition system (LTS) model and a set of properties (specified using fluent linear temporal logic “FLTL”). Finally,
the LTS model and the properties are fed to the LTSA tool for performing the validation. Also, there are a number of approaches that have been introduced to validate adaptive software systems (e.g. [5-6]).

To validate a service’s requirements using the above approaches, the engineer is responsible for enumerating and specifying the service variants that suit different context situations. But, for a complex service, this task is tedious and error prone, where the service’s high variability introduces an explosion of the possible service variants [5]. To tackle this challenge, in our approach, the engineer specifies a small number of functional and adaptation scenarios. Then, a large number of the service variant specifications are generated from the scenarios and checked against the relevant service properties automatically.

(4) Checking the Adaptation Requirements’ Consistency: Few efforts have been proposed to check the consistency of a service’s adaptation requirements. Sama et al. introduced an approach to validate the adaptation requirements (captured as a set of rules) of mobile applications [21]. They introduced a set of algorithms to identify rules that are fired incorrectly or correct rules that are not triggered because of inaccurate context information. There are also a number of general approaches for analysing condition-action rules (e.g. [22]) to detect unexpected behaviour of the rules (e.g. rules may mutually trigger one another that leads to unexpected rules execution). These approaches only ensure that the rules are fired correctly. However, the set of adaptation actions that are fired in response to context changes by these rules may be inconsistent with each other. To detect this inconsistency, in our approach, we identify and check possible adaptation actions (in scripts) to ensure that only valid actions are applied into the service at runtime.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a scenario-based approach to assist the software engineer in validating the varying requirements of context-aware adaptive services. Compared to existing approaches, our approach has the following key contributions. First, a service’s properties are specified graphically in a form similar to the scenarios. Thus, the properties can be clearly captured and easily specified. Second, our approach enumerates and generates the service variants form the service scenarios, which are then validated against the service properties automatically. It also checks the consistency of the service’s adaptation requirements.

As future work, we plan to extend our approach to support the runtime evolution and incremental validation of context-aware adaptive services (initial results can be found in [23]). As our case studies have produced promising results, we also plan to carry out further evaluations to assess the approach’s applicability.

ACKNOWLEDGMENT

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

REFERENCES