Enabling Runtime Evolution of 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, acolmans}@swin.edu.au

Abstract—There is an increasing demand for context-aware adaptive services that can evolve at runtime in response to unanticipated changes in their environments or functionalities. Enabling the runtime evolution of such context-aware adaptive services is still a major challenge. In this paper, we introduce a novel approach to tackle this challenge. At the development time, our approach explicitly models a context-aware adaptive service from three aspects: functionality, context, and adaptive behaviour. As such, these aspects and their relationships can be clearly captured and easily manipulated. The approach also generates the executable artifacts of the composite service from their models. These artifacts are engineered with the ability to be changed at runtime (i.e. evolvable artifacts). To cope with unanticipated changes, we adopt the models@runtime concept for keeping the service model alive at runtime. This model is then manipulated by the software engineer to take into account such changes. To apply the model changes to the running service’s composition, differences between the running service model and its evolved model are computed. Then, adaptation actions corresponding to the differences are generated and applied to the service’s evolvable artifacts. To demonstrate the approach applicability, we have used it to develop and evolve two case studies: travel guide and electronic exam services.

Keywords—Runtime Evolution, Models@runtime, Context-awareness, Self-adaptation.

I. INTRODUCTION

There is an increasing demand for context-aware adaptive services that have the ability to evolve at runtime in response to unanticipated changes [1]. These changes can be in their environments or functionalities where (a) the services are usually deployed into environments that are not totally anticipated at the development time [2]; (b) the users may need new functionalities or the provider wants to enhance the services with new features [3].

In the past few years, a number of approaches have been introduced to enable a service’s runtime adaptation to cope with context changes (e.g. [4-6]). However, these approaches only support the service adaptation in response to changes that have been anticipated during its development time. To cope with the unanticipated changes, a number of techniques have been introduced (e.g. [7-10]). However, two major challenges still remain. First, the context information that is needed by a context-aware service (to operate effectively or to trigger its adaptation) is intertwined with its functionality or management. In addition, the relationships between the service composition elements are not explicitly represented. Therefore, the service model cannot be clearly captured and it becomes difficult to evolve when unanticipated changes occur. Second, the runtime changes to a composite service are specified directly into the service’s source code (e.g. [7]), or as adaptation actions (in a textual format) that need to be applied into the service (e.g. [9]). Consequently, in a complex service, the task of “specifying runtime changes to the service” is tedious and error prone.

To tackle these challenges, in this paper, we introduce a novel approach to enabling the runtime evolution of context-aware adaptive services. In developing such services, our approach explicitly models a service composition from three aspects: functionality, context, and adaptive behaviour. As such, these aspects and their relationships can be clearly captured and easily manipulated at runtime. The executable artifacts of the composite service are then generated from their models. These artifacts have the ability to change at runtime (i.e. evolvable artifacts), where they are engineered with a set of adaptation actions such as adding, removing, or modifying their sub-elements. To evolve a running context-aware adaptive service in response to unanticipated changes, we adopt models@runtime concept where the service model is kept alive at runtime [11]. Then, the runtime model of the service is manipulated (evolved) by the software engineer to take the unanticipated changes into account. To apply the model changes to the running composite service, differences between the running service model and its evolved model are computed. Then, a set of adaptation actions (as a script) that are corresponding to the differences are generated and applied to the service’s evolvable artifacts.

A graphical tool has been developed to support the runtime evolution of context-aware adaptive services. To demonstrate the applicability of our approach, we have used it to develop and evolve two case studies: travel guide and electronic exam services.

The contribution of the paper is twofold: (1) a technique to modelling a context-aware adaptive service from three aspects (i.e. functionality, context, and adaptive behaviour) explicitly, so that these aspects and their relationships can be clearly captured and easily evolved; (2) a technique to evolving a running composite service by manipulating the service’s runtime model and reflecting the model changes to the service automatically.

The remainder of the paper is organized as follows. In Section 2, we start by introducing a motivating scenario and an overview of the approach. In Sections 3 and 4, we present our approach to developing and evolving context-aware adaptive services. The application of our approach to two case studies is discussed in Section 5. Section 6 analyzes existing work with respect to our approach. We conclude the paper in Section 7.
II. MOTIVATING SCENARIO AND THE APPROACH OVERVIEW

In this section, we present a motivating scenario and give an overview of the approach.

A. Motivating Scenario

Let us consider a travel guide service that is required to obtain/sense the available context information (e.g. the route preferences and the traffic information), and select a suitable route planner that takes the context information into account. After that, the route planner shows the available routes to the driver based on her destination and current context (e.g. her location, her route preferences, and the traffic information). Finally, the driver selects a route and starts the journey.

While the travel guide service is in operation, (1) a new type of context information source (i.e. the area speed limit) becomes available, and the service provider wants to use it to alert the driver when her vehicle speed exceeds the speed limit; (2) the travel guide provider wants to enhance the service functionality by adding an attractions finder function, so that the driver can use this function to find attractions in a route suggested by the route planner and select some of them for visit. The service should also be available 24/7 (24 hours a day, 7 days a week) to increase users’ satisfaction. As such, the travel guide service needs to evolve at runtime to take into account new context information (e.g. the speed limit) or new functionality (e.g. the attractions finder).

B. The Approach Overview

To evolve a running context-aware adaptive service, the service needs to be developed with this feature in mind. Thus, our approach has two main phases: development and evolution as shown in Figure 1. At the development phase, the service requirements are used for designing the service model. This model captures the service’s functionality, context, and adaptive behaviour (Step 1). The service model is then transformed to an executable service. In particular, the generated runtime artifacts of the executable service are engineered with the ability to change at runtime (Step 2).

When the service is in operation, the service manager changes the service roles, role players’ bindings, and their relationships to maintain the service viability in the face of the environment changes. These differences are then used for generating a set of adaptation actions which are applied to the running service composition (Step 4).

III. DEVELOPING EVOLVABLE SERVICES

To enable a service’s runtime evolution, the service’s runtime artifacts need to be evolvable. In this section, we describe a technique to designing an evolvable context-aware adaptive service (Step 1), and discuss how to generate the service’s evolvable artifacts from its design model (Step 2).

A. Designing Evolvable Context-aware Adaptive Services

To design context-aware adaptive services that are able to evolve at runtime (i.e. Step 1), we follow an organizational approach [12]. We adopt such approach because it represents relationships between a service’s elements explicitly, so that the elements and their relationships (shown in Figure 2) can be clearly captured. It also keeps the service model alive at runtime, so that the service can be easily evolved.

A service as an organisation is a set of dynamic relationships between its roles to maintain the service viability in a changing environment [12]. The relationships (defined as contracts) specify the permissible interactions between the service’s roles (i.e. what tasks a role can request from others). Thus, they are used to specify the “position descriptions” of the service’s roles. The position descriptions specify what tasks the service’s roles should do, while there are players (specific partner services) who actually perform the tasks by playing these roles. In addition, to coordinate the interactions between the service’s roles to achieve composite tasks, a set of processes are specified. These processes define in what order a set of simple tasks are performed to achieve composite tasks. Furthermore, in response to environment changes, the service manager changes the service roles, role players’ bindings, and their relationships to maintain the service viability in the face of the environment changes.

Following the organizational approach, a meta-model for an evolvable context-aware adaptive service is presented in Figure 2. The service composition consists of two main sub-composites: functional and management. In the following, we describe these two composites in detail.

1) The Functional Composite: To capture the service’s functionality, the functional composite (as shown in Figure 2) has a set of functional roles that interact with each other through functional contracts (i.e. the functional structure). In addition, it has a set of behaviour processes to capture the service’s functional behaviour. Furthermore, to make the service’s functionality context-aware, the composite has a set of context roles bound to context providers to make the context information available (i.e. the context model).

1.1 The Functional Structure: The service’s functional structure is modelled as a set of functional roles that interact with each other through functional contracts (see Figure 2), so that the service roles and their relationships can be easily changed. In addition, each role can be played by one or more players, where the role players’ binding can be modified at runtime. To facilitate stakeholder involvement in capturing the service structure, we have introduced intuitive notations to represent the meta-model concepts as shown in Figure 3.

Figure 1. A process to developing and evolving a context-aware service

At the evolution phase, first, the service’s runtime model is manipulated to cope with the unanticipated changes (Step 3). Second, differences between the running service’s model and its evolved model are computed. These differences are then used for generating a set of adaptation actions which are applied to the running service composition (Step 4).

Figure 2. The service composition consists of two main sub-composites: functional and management. In the following, we describe these two composites in detail.
1.1.1 Functional Roles and their Players: The functional roles represent the service functionality, where each role is an abstract definition of the tasks (functions) that this role should provide and there are one or more players to provide its functionality at runtime. For example, there are two route planning algorithms that can play the route planner role (see Figure 3). RoutePlanner1 considers the traffic information in calculating the routes, while RoutePlanner2 do not.

1.1.2 Functional Contracts: The functional contracts are used for capturing the interactions between the functional roles. Each contract has the following items. First, each contract has an identifier and it exists between two functional roles (i.e. roles A and B as shown in Listing 1). For example, the contract “FC2” exists between the user (role A) and route planner (role B) roles as shown in Figure 3. Second, the contract has a set of permissible interactions between the contracted roles as shown in Listing 1. Each interaction has (1) an identifier (e.g. “i2”); (2) an operation that needs to be performed by requesting that interaction and the operation has a name (e.g. PlanRoutes2) and a set of input parameters (e.g. destination, current location, and traffic information); (3) a direction to specify who is responsible for providing the operation included in that interaction (e.g. “AtoB” which means the route planner role is responsible for providing the route calculation operation); (4) a return type (e.g. Routes).

Listing 1. Part of the functional contract “FC2”

<table>
<thead>
<tr>
<th>Functional Contract ID</th>
<th>FC2: User_RoutePlanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parties: Role A</td>
<td>User;</td>
</tr>
<tr>
<td>Role B</td>
<td>RoutePlanner;</td>
</tr>
</tbody>
</table>

Interaction Clauses:

\[ i1: [\text{PlanRoute1} (\text{Destination}, \text{CurrentLocation}), \text{AtoB}, \text{Routes}]; \]

\[ i2: [\text{PlanRoute2} (\text{Destination}, \text{CurrentLocation}, \text{TrafficInformation}), \text{AtoB}, \text{Routes}]; \]

1.2 The Functional Behaviour: The service’s behaviour specifies how the service behaves at runtime to provide its functionality. In response to a user request, the service executes a single task or a set of tasks (i.e. a composite task). The request of a single task is supported directly through the functional contracts. To support the request of composite tasks, a set of behavior processes are specified within the service composition (see Figure 2).

In designing the processes, we follow an event-based approach [13]. We adopt the event-based approach because it represents a process as a set of loosely coupled tasks that are related to each other through pre (events that enable the execution of a task) and post (events that are generated upon a task completion) conditions as a set of events (see Figure 4), so that the process can be easily evolved by changing the pre and post events of its tasks. The process’s tasks can be simple or composite tasks. A simple task corresponds to a single function execution, while a composite task is a process itself. Similarly, the pre and post conditions of a task can be simple (single event) or complex (a set of events that are combined with each other using logical operators such “and” and “or” as shown in Figure 4) events. The process also has two events that specify its start and end.

To simplify the Figure, we have removed associations between the service’s roles and contracts, and they are described in the text only.
the traffic information availability, a suitable route planning function is selected. Then, a set of routes are suggested to the user where she can select a route.

1.3.2 Contextual Contracts: A contextual contract defines the context information that is required by a functional role. For example, the contextual contract “CC2” shown in Listing 2 specifies that the route planner role needs to know the live traffic information, so that the route planner can calculate the routes effectively.

```
Listing 2: The contextual contract “CC2”

Contextual Contract ID: CC2; TrafficInfo_RoutePlanner
Parties: Context Source: TrafficInfo;
Context Consumer: RoutePlanner;
Context Attributes: a1: TrafficInformation;
```

2) The Management Composite: To specify a service’s adaptation in response to anticipated context changes (i.e. its adaptive behaviour), we have introduced the management composite (see Figure 2). This composite has three types of roles (i.e. functional service, context, and management), and two types of contracts (i.e. management and contextual) to capture the relationships between the composite roles. In the following, we discuss the management composite’s elements except the contextual contract which is same as the contextual contract in the functional composite.

2.1 Functional Service and Context Roles: The functional service role represents the functional composite that needs to be adapted at runtime (e.g. the travel guide service shown in Figure 5) while the context roles capture context information that triggers the service’s runtime adaptation (e.g. the user information as shown in Figure 5).

2.2 Management Role and its Player: The management (organizer) role bound with its player decides the required adaptations in response to anticipated context changes. We model the organizer player as a set of Event-Condition-Action rules [14]. We adopt the rule-based approach to capture the service’s reactions to context changes because of its expressiveness and availability of the tool support. A rule event is usually a context change where the service needs to adapt itself in response to such change. The rule condition specifies a context situation that needs a reaction(s). The rule action is a set of adaptation actions in response to the context changes. In general, the adaptation actions are to add, remove, or modify a service element. For example, to change the service roles, we have three adaptation actions: add role, remove role, and change role’s player binding. In the same manner, we have adaptation actions for adding, removing, or modifying a functional contract, a contextual contract, and a behaviour process. An example adaptation rule is shown in Listing 3. This rule is activated (i.e. event) when the traffic information is not available (i.e. condition). In response to this change, the service is adapted (i.e. action) by removing the contextual contract “CC2”, binding the route planner role with the player “RoutePlanner2”, etc (see Listing 3).

```
Listing 3: A rule to cope with the unavailability of the traffic information

Rule “AdaptationRule1”:

When: ValueChanges (TrafficInfoAvailability),
if TrafficInfoAvailability = False;
```

2.3 Management Contracts: A set of adaptation actions need to be applied to the functional composite in response to anticipated context changes. Thus, a management contract between the functional service role and the organizer role exists (e.g. the contract “MC1” in Figure 5). This contract defines adaptation actions that need to be performed into the service functionality. An example management contract is “MC1” shown in Listing 4.

```
Listing 4. Part of the management contract “MC1”

Management Contract ID: MC1; TravelGuideService Organizer
Parties: Role A: ServiceOganizer; Role B: TravelGuideService;
Management Actions:
m1: [removeContract (“CC2”), AtoB, Boolean];
m2: [removeRole (“TrafficInfo”), AtoB, Boolean];
```

B. Realizing Evolvable Context-aware Adaptive Services

In our previous work, we have introduced the ROAD framework which is an extension to Apache Axis2 to realize adaptive services [15]. To realize a context-aware adaptive service using this framework (i.e. Step 2), we transform the service’s design model to a model that is compatible with the ROAD framework. The major transformations involved are described below and further details can be found in [16].

First, in the ROAD model, the service’s functionality and management are captured as one composite. But, we provide an extended view of the service management as a separate composite. Therefore, we combine the two composites of a service designed by our approach to a ROAD composite.

Second, the context information is maintained as a set of facts in the ROAD model. Each fact contains one or more
context attributes. The facts can be provided or consumed by the service roles. In our model, each context role description is a collection of context attributes that describes a context entity. This creates a correspondence between a context role in our model and a fact in the ROAD model. As such, we transform each context role to a fact in the ROAD model.

Third, to enable the execution of the adaptation rules, we transform them to Drools' rules. In transforming a rule, we use the Drools rule “When” part to specify the rule event and the rule “Then” part is used for capturing both the condition and the action of the rule. We also generate a Java class that represents the organizer player. This class has a method that uses the generated rules to decide adaptation actions that are then applied to the running service in response to anticipated context changes. It also maintains a runtime representation of the service model using the models@runtime concept [11]. This model can be manipulated at runtime to cope with the unanticipated changes (as discussed in Section 4).

The above transformation process generates a model that is compatible with the ROAD framework. When this model is deployed to the ROAD runtime environment, an instance of the service is created. This instance contains the service’s roles, contracts, and players. It also supports the application of different adaptation actions to that service. For example, the deployed service composition is able to add, remove, or modify its roles and contracts (i.e. evolvable composition). Similarly, each deployed service artifact is evolvable where it has the ability to add, remove, and modify its sub-elements (e.g. adding an interaction to a functional contract).

IV. RUNTIME EVOLUTION OF CONTEXT-AWARE SERVICES

To incorporate new context information or functionality into a running service composition with our approach, the software engineer manipulates the service’s runtime model to cope with these unanticipated changes (i.e. Step 3). Then, the changes to the runtime model are automatically applied to the running service (i.e. Step 4).

A. Manipulating the Service Runtime Model

To cope with unanticipated changes, the service runtime model (maintained in the organizer player) is modified by the software engineer. First, in response to unanticipated changes in the service’s environment, a context attribute or a context entity is added to the service model. To add a context attribute, an already specified contextual contract is modified to include such attribute. For example, to take into account the area speed limit, the contextual contract “CC2” (shown in Listing 2) is changed by adding this context attribute, so that the route planner becomes able to get the speed limit from the traffic information provider (see Figure 6). To add a context entity, three elements are added to the service’s runtime model: a contextual contract, a context role, and a context provider(s). For example, to add the weather information, the contextual contract “CC3” is added to specify that the attractions locator role needs to know the weather conditions. In addition, to make such context information available, the weather role and its player (i.e. the weather service) are added to the service’s runtime model as shown in Figure 6.

Second, to add a new functionality, a number of changes are made to the service’s runtime model. A functional role and its player(s) are added to provide this functionality. Then, one or more functional contracts are added, so that the new functionality becomes able to interact with the service elements. Finally, a behaviour process is added to specify a sequence of interactions that allow the user to use the new functionality. For example, to include the attractions finder function, the functional composite model is changed (as shown Figure 6) by adding the attractions locator role and its player. In addition, the functional contract “FC3” is added to allow the user interactions with this new function, and a behaviour process is added to specify the sequence of interactions to be followed by the user to find attractions.

Third, not only the service functionality and its context model evolve (as discussed above), but also its adaptive behaviour evolves. To include new adaptive behaviours, a set of adaptation rules are added to capture these adaptive behaviours. For example, to make the user able to include the attractions finder function in his service, a rule is added. This rule is activated when the user wants to add such function. In response to this change in the user needs, the service adapts its functionality and context model by adding a set of roles (e.g. the attractions locator and weather roles), adding a set of contracts (e.g. “FC3” and “CC3”), and binding players and context providers with the added roles (e.g. bind the attractions locator role with its player), etc.

Fourth, the runtime model of the service may change by removing or modifying its elements. For example, the service provider may remove a service function, modify an adaptation rule, remove a context entity, etc.

B. Applying the Model Changes to the Running Service

To apply the changes of the service’s runtime model to the running service, we compute the differences between the service’s evolved model and its initial model, and generate adaptation actions that are corresponding to the differences as an evolution script. Then, these actions are applied to the service’s runtime artifacts. First, to compute the differences between the service’s initial model and its evolved model
and generate their corresponding actions, we parse the two models to identify elements that are added, removed, or modified. Then, for each model change, an adaptation action realizing that change is generated. Due to space constraint, we only show a part of the algorithm we use to compute the models’ differences in Listing 5. This part is used for identifying changes in the service contracts. We first get the contracts in the old and new models of the service (Lines 1 and 2). After that, iterating over them finds the added contracts (i.e., contracts that do not exist in the initial service model). If a contract is new, adaptation actions to add this new contract are inserted to the evolution script (Lines 10 to 15). In the same manner, removed and modified contracts can be identified as shown in Listing 5.

Listing 5. Part of the algorithm to identify changes in a service’s contracts

```java
void identifyChangedFunctionalContracts(ServiceModel RM, ServiceModel EM, EvolutionScript ES) {
    // RM: Initial (deployed) service model
    // EM: Evolved model
    1. List oldContracts = RM.getContracts();
    2. List newContracts = EM.getContracts();
    // Identify added contracts
    3. FOR each FunctionalContract fc in newContracts
        4. Boolean exist = false;
        5. FOR each FunctionalContract fc1 in oldContracts
            6. IF fc.getId() equals fc1.getId() THEN
                exist = true;
                7. ELSE
                    ES.addAction("organizer.addBehaviorToProcess(" + fc.getId() + ", " + fc.getDescription() + ", " + fc.getRoleA() + ", " + fc.getRoleB() + ");
                    break;
                END IF
            8. END FOR
        9. END IF
    10. IF exist == false THEN // the contract is new where it does not exist
        11. ES.addAction("organizer.addContract(" + fc.getId() + ", " + fc.getDescription() + ", " + fc.getRoleA() + ", " + fc.getRoleB() + ");
        12. END FOR
    13. END FOR
    // Identify removed contracts
    14. FOR each FunctionalContract fc in oldContracts
        15. Boolean stillExist = true;
            FOR each FunctionalContract fc1 in newContracts
                16. IF fc1.getId() equals to fc.getId() THEN
                    17. stillExist = false;
                    18. END IF
            19. END FOR
        20. END IF
        21. IF stillExist == false THEN // the contract is not exist anymore
            22. ES.addAction("organizer.removeContract(" + fc.getId() + ");
        23. END IF
    24. END FOR
    // Identify modified contracts
    25. FOR each FunctionalContract fc in newContracts
        26. FOR each FunctionalContract fc1 in oldContracts
            27. IF fc1.getId() equals fc.getId() THEN
                28. END IF
                29. END FOR
        30. END FOR
    31. END FOR
    // Checking added functional interactions
    32. FOR each FunctionalContract fc in newContracts
        33. ES.addAction("organizer.addInteraction(" + fc.getId() + ");
    34. END FOR
    // Checking removed functional interactions
    35. ES.addAction("organizer.removeInteraction(" + fc.getId() + ");
    36. END
}
```

An example evolution script generated using our algorithm to evolve the travel guide service is shown in Listing 6. This script includes part of the adaptation actions corresponding to the differences between the models shown in Figures 3 and 6. It is worth noting that the generated actions are in the form of executable Java code, so that they can be executed at runtime to evolve the running service.

Listing 6. Part of the evolution script to evolve the travel guide service

```java
organizer.addNewRole("FR4", "AttractionsLocator");
organizer.addNewContract("FC3", "User-Attractions.locator");
organizer.addNewTerm("I1", "FindAttractions", "void");
organizer.addNewRole("FR4", "AttractionsFinder");
organizer.addNewFact("TrafficInfo", "SpeedLimit");
organizer.changePlayerBinding("FR4", "http://localhost:8080/axis2/services/AttractionsFinder");
organizer.addBehaviorTerm("P2_BI1", "NotifySpeed", "void");
organizer.addBehaviorToProcess("P2_BI1", "AttractionsPrefRequested", "AttractionsPrefSelected");
organizer.addBehaviorToProcess("P2_BI1", "AttractionsPrefRequested", "AttractionsPrefAvailable");
organizer.addBehaviorToProcess("P2", "P2_BI1", "AttractionsPrefRequested");

Second, to execute the generated evolution script, we have used Jaavassist4 to create a Java method on-the-fly. This method has a reference to the running service, and the Java code that corresponds to the differences between the service’s new and old models (i.e., the evolution script). It also communicates with the running composite service to perform required changes by executing adaptation actions that are engineered into the service at the development time (see Section 3.B). To execute this method, we have created a Java class with the evolve method that has an empty body at the development time. At runtime, we modify this method by replacing its body with a new code that has adaptation actions to be executed. Finally, we create an instance of that class, and use it to execute the evolve method. More details about computing the models’ differences, generating the evolution script, and executing the generated script can be found in [16].

C. Tool Support

We have developed a tool that supports the software engineer in performing the following tasks (see Figure 7 for a screenshot). First, the engineer can use our tool to design a context-aware adaptive service (Step 1). Second, the tool enables an evolvable service’s realization by transforming the service model designed by our approach to a ROAD model as discussed in Section 3.B (Step 2). Third, to change a running service’s model in response to the unanticipated changes, the software engineer can use our tool to load the service’s runtime model and to perform the required

---

4 In the same manner, changes to the service’s roles, functional behaviour, and adaptive behaviour are identified
5 We assure the consistency of an evolution script by taking into account the actions’ dependences in the script generation process. For example, a contract should be added before adding interactions to it as shown in Lines 10-15 of Listing 5.

http://www.cs.g.is.titech.ac.jp/~chiba/javassist/
changes to that model as described in Section 4.A (Step 3). Finally, the tool is able to compute the differences between the old and new models of a running service, generate an evolution script, and execute the script to evolve the running service as discussed in Section 4.B (Step 4).

![Figure 7. Screenshot from the developed tool](image)

V. CASE STUDIES

In this section, we demonstrate the approach applicability by using it to develop and evolve two case studies: travel guide and electronic exam services (further details can be found in [16]). The results of the case studies show that with our approach a service’s functionality, context, and adaptive behaviour can be clearly captured, and easily evolved to cope with the unanticipated changes.

A. The Travel Guide Service

We have used our approach to develop and evolve the travel guide service described in Section 2. At the design time, the functional composite has 3 functional roles that are connected by 2 functional contracts (see Figure 3). These roles have 4 players. In addition, there are 2 context roles connected to the functional roles with 2 contextual contracts, and 3 behaviour processes. The management composite has 2 context roles, 2 contextual contracts, and 10 adaptation rules. At runtime, to cope with the unanticipated changes, the service’s runtime model is changed by adding a functional role and its player, a functional contract, 3 context roles connected with the service roles by 3 contextual contracts, a behaviour process, and 6 adaptation rules (see Table 1). Using our approach these model changes are identified, and applied to the running service automatically.

**TABLE I. SUMMARY OF THE CASE STUDIES**

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Travel Guide</th>
<th>E-exam</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initial</td>
<td>Evolved</td>
</tr>
<tr>
<td>FC’s Functional Roles/Players</td>
<td>3/4</td>
<td>4/5</td>
</tr>
<tr>
<td>FC’s Functional Contextual Contracts</td>
<td>2/2</td>
<td>3/4</td>
</tr>
<tr>
<td>FC’s Context Roles/Providers</td>
<td>2/3</td>
<td>4/5</td>
</tr>
<tr>
<td>FC’s Behaviour Processes</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>MC’s Context Roles/Providers</td>
<td>2/2</td>
<td>3/3</td>
</tr>
<tr>
<td>MC’s Contextual Contracts</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>MC’s Adaptation Rules</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

B. The Electronic Exam Service

We have also used our approach to develop and evolve a larger service: an “electronic exam (e-exam) management service” based on the example in [17]. 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, have the ability to be adapted at runtime based on its users’ context (e.g. a student is only able to take an exam when he is at the exam location), and can evolve at runtime to include new features and to take into account new context information. Table 1 summarises the number of elements in the service model and the changes to these elements to cope with the unanticipated changes.

VI. RELATED WORK

Recently, a number of approaches have been proposed to support the development of services that have the ability to change at runtime. These approaches are either concerned with adapting a service in response to changes that are anticipated at the design time or focused on enabling runtime changes to a service in response to unanticipated changes. In this section, we discuss related work from these two angles.

A. Handling the Anticipated Context Changes

CEVICHE is a framework that combines complex event processing and business process adaptation to enable runtime changes to a service [4]. It has a language called SBPL (Standard Business Process Language). This language is an extension to the BPEL to make business processes more flexible by defining adaptation points in a business process during its design time. Wang et al. introduced an approach to adapt a composite service in response to runtime context changes, while achieving the optimal service composition by reinforcement learning [5]. In their approach, they model the service composition as a Markov Decision Process, so that multiple alternative workflows can be incorporated into that composition at the design time. At runtime, the optimal workflow that has the best quality of service is selected using reinforcement learning. Apto is an approach to developing adaptive service-based processes [6]. To capture a process’s adaptive behaviour, they introduced the term “adaptation fragment” that specifies the required adaptation to a process to cope with context changes. At the deployment time of the process, the environment context is acquired. Then, based on current context situation, an adaptation fragment is selected and applied to the process to suit the context situation.

These approaches enable runtime changes to a service in response to context changes. However, the runtime changes are limited to adaptations that are specified at the design time to cope with the anticipated context changes. Thus, a service developed by these approaches cannot be changed at runtime to cope with the unanticipated changes. In our approach, we specify a service’s adaptive behaviour at the design time. This behaviour is then used at runtime to adapt the service in response to the anticipated context changes. In addition, the service’s runtime artifacts (generated by our approach) are evolvable, where they are engineered with general adaptation actions to add, remove, and modify their sub-elements.

---

7 Functional Composite
8 Management Composite
B. Enabling the Runtime Evolution of a Service

A few efforts have been introduced to support a service’s runtime evolution to cope with the unanticipated changes. Oreizy et al. proposed an architecture-based approach for runtime service (software) evolution [9]. To effectively change a running service, they maintain an instance of the service architecture at runtime. The service adaptation in response to the unanticipated changes is then specified at the architecture level using a set of actions such as add, remove, or replace a service’s element as a script. Finally, the script is executed to evolve the running service. Another approach is introduced by Fang et al. to enable runtime changes to an instance of a BPEL process [10]. These changes are limited to skip or to retry an activity in the enacted process. Yu et al. introduced an approach to enable model-driven development of adaptive services [18]. They enable runtime changes of a service’s adaptive behaviour. These changes are performed manually by the service developer on the service’s source code. Also, there are a number of approaches that support a service’s runtime update in general (e.g. [7-8]).

In these approaches, first, the context information that is needed by the service is intertwined with its functionality or adaptive behaviour. Also, relationships between the service elements are not explicitly represented. As such, the service model cannot be clearly captured, and the runtime evolution of the service aspects to cope with the unanticipated changes becomes difficult. In our approach, we explicitly model the service from three aspects (i.e. functionality, context, and adaptive behaviour), and maintain a model of the service at runtime. Thus, the service elements and their relationships can be clearly captured and easily evolved.

Second, to cope with the unanticipated changes, runtime changes to a service are specified directly into the service’s source code (e.g. [7]), or specified textually as adaptation actions that need to be applied to the service (e.g. [9]). Thus, in a complex service, this task (i.e. specifying the service’s runtime changes) is tedious and error prone. To ease this task, with our approach the software engineer can load a service’s runtime model, and manipulate this model to cope with the unanticipated changes. Finally, the service’s model changes are applied automatically to its evolvable artifacts.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced an approach to enable the runtime evolution of context-aware adaptive services. Compared to the existing approaches, our approach has the following key contributions. First, our approach explicitly captures a service’s model from three aspects: functionality, context, and adaptive behaviour. In addition, we adopt the models@runtime concept to maintain a runtime model of the service while it is in operation. Thus, the service aspects and their relationships can be clearly captured and easily changed at runtime. Second, to ease a service’s runtime evolution in response to the unanticipated changes, with our approach, the software engineer can perform the service’s runtime changes at the modelling level. These changes are then automatically applied to the running service composition.

As future work, we plan to investigate how to capture a service’s requirements and generate the service’s design model from its requirements (initial results can be found in [19]). We also plan to perform more evaluations to further assess the approach applicability.

ACKNOWLEDGMENT

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

REFERENCES