Guiding Architects in Selecting Architectural Evolution Alternatives

Selim Ciraci[1], Hasan Sözer[2], and Mehmet Aksit[3]

[1] Pacific Northwest National Lab. Richland, WA, USA
selim.ciraci@pnl.gov
[2] Özyeğin University, İstanbul, Turkey
hasan.sozer@ozyegin.edu.tr
[3] University of Twente, Enschede, The Netherlands
m.aksit@ewi.utwente.nl

Abstract. Although there exist methods and tools to support architecture evolution, the derivation and evaluation of alternative evolution paths are realized manually. In this paper, we introduce an approach, where architecture specification is converted to a graph representation. Based on this representation, we automatically generate possible evolution paths, evaluate quality attributes for different architectural configurations, and optimize the selection of a particular path accordingly. We illustrate our approach by modeling the software architecture evolution of a crisis management system.

1 Introduction

To add new features, to adopt new technologies, or to improve quality factors, software systems evolve [1]. Usually there exist multiple evolution alternatives, which should be evaluated rigorously at a high abstraction level [2]. In architecture evaluation methods like ATAM [3], trade-off analysis is a manual process. Tools and techniques are developed to guide the architects in planning and carrying out architecture evolution [2, 4, 5]. These, however, either do not support trade-off analysis or require the manual generation of evolution alternatives.

In this paper, we propose a method and the accompanying tool set to aid the user in selecting the "best" evolution alternative. Hereby, the architectural changes (anticipated/common evolutions) are modeled in xADL [6] in ArchStudio [7], with proposed extensions that specify how the architectural elements are changed (e.g., added, removed) and their impact on quality attributes. Our tools convert these specifications to graph transformation rules and store them in a repository. The user specifies the desired types of evolutions and inputs the initial architecture (also specified in xADL). Our tools, first, convert the initial architecture to a graph and, then, fetch architectural changes (from the repository) whose categories match the desired evolutions. Next, the evolution alternatives are generated with a graph transformation tool. Quality attributes are evaluated for each evolution alternative based on the impact of different

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 252–260, 2011.

architectural changes. Finally, the user can query and score the evolution alternatives according to the desired structural and path properties. An optimizer uses these scores and quality attributes to select the best evolution alternative. We illustrate our approach by modeling the software architecture evolution of a crisis management system.

This paper is organized as follows: next section introduces a motivating example, which will be used throughout the paper for illustration purposes. Section 3 explains the modeling of architectural changes. Section 4 explains the generation of the architectural alternatives and the selection of the optimal evolution alternative. Section 5 discusses the case study. Finally, conclusions are provided in Section 6.

2 Motivating Example

A crisis management system (CMS) [8] is used for managing the resources (e.g., ambulances) to aid in crisis situations. To support coordination of crisis resolution processes and efficient allocation of resources, the CMS architecture design incorporates programmable crisis managers called *scenarios* and resource allocation strategies. Figure 1 shows the architecture design of the CMS with only one scenario, the car crash scenario, and one allocation strategy, i.e., first come first serve (FCFS). In principle, there can be many scenarios and allocation strategies implemented as separate components at the corresponding layers. The users or the mobile devices communicate with the CMS through the *Crisis Manager* component. The *ScenarioCtrl* interface is used for relaying requests and events to the connector *Scenario Control Interface*. This connector decodes and forwards the received events to corresponding scenario(s) through interface dedicated for different types of outside events. Scenarios allocate resources through the connector *ResourceManager*. This connector sends resourse allocation messages to components representing the resource allocation strategy. The resource allocation strategy finds the best candidate resource from the list of resources and grants the allocation.

Anticipated Type of Evolutions: Anticipated evolutions for CMS include the addition of new scenarios and resource allocation strategies. These evolutions can be incorporated to the architecture in different ways. For example, assume that we want to add a new scenario called *presidential emergency*. This scenario has

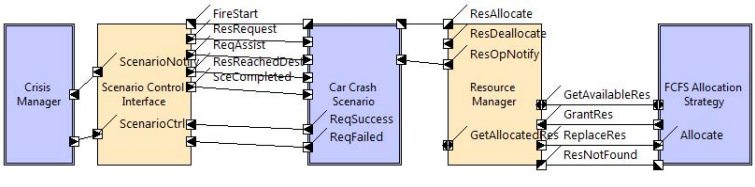

Fig. 1. A (partial) CMS architecture design.

a higher priority; so when a report about a presidential emergency comes and there are no resources available, an ongoing car crash scenario needs deallocate resources. We can add this scenario in two ways: *i)* We can extend connector *ResourceManager* to receive an outside event that causes other scenarios to release resources, *ii)* we can add a new resource allocation strategy. In the former case some of the resource allocation tasks are handled by the scenarios which might not be desirable. The later might be more costly as a new allocation strategy needs to be implemented. Such alternatives together with their pros and cons must be studied not to implement a suboptimal or undesired change.

3 Preparation Steps

As preparations steps, architectural changes are modeled. These are anticipated/repeating changes that are specific to the architecture of the software system at hand. We use the *Arch-Studio* tool [7] to describe such changes in *xADL* [6]. Hereby, changes are specified as partial structures annotated with operations, i.e., add/remove elements. Our tool set converts these specifications to the corresponding graph transformation rules. In this way, the graph system and the underlying theory are hidden from the users.

An architectural change that is represented as a graph transformation rule has a left-hand side (LHS), a right-hand side (RHS), a name, and a type. Therefore, each architectural change is specified in 4 steps. First, a name for the change is specified. Second, the architectural elements required by the change is modeled; this is analogous to the LHS of a transformation rule, e.g., if we want to add a new interface to the connector *Scenario Control Interface*, this connector is required and as such should be modeled in the architecture change model with the same name to be able to apply the change. Third, the transformation executed by the architectural change is modeled using operators that add/remove elements. An architectural element whose name starts with *new:* is added, *delete:* is removed, *not:* should not be in the architecture for the change to be applied. The elements with operators, and the elements from the second step forms the RHS of the transformation rule. For example, the architectural change *AddEventFirePreEmpt* adds a new interface called *FirePreEmpt*, which is modeled by the interface *new:FirePreEmpt* in Figure 2. We do not need to apply this change, if the connector *Scenario Control Interface* has already an interface named *FirePreEmpt*. We model this with the interface *not:FirePreEmpt*.

In the forth (last) step of specifying an architectural change, type of the architectural change is defined. This type is specified as a set of keywords describing what the architectural change does. It is possible to have more than one change for a type. Once the architectural change is modeled, it is stored in the repository managed by the CDE tool [9]. CDE tool first converts the change modeled in xADL to graph-transformation rules

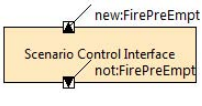

Fig. 2. An architectural change

and, then, indexes it according to the specified type (the tools described in this paper can be located at http://sourceforge.net/projects/caae/).

Quality Attributes: In order to capture these quality attributes in architectural changes and models, we have extended the existing xADL schema with new types; namely reliability interface and cost component. The concept of an interface with analytical parameters has been borrowed from [10]. The reliability interface is added to components/connectors and it consists of a reliability value R, where $0 < R < 1$. The cost component is a special component to which interfaces can not be added and which is named **Cost**.

Reuse of Architectural Changes: To promote reuse of architectural changes, our approach supports parameterization of names/descriptions. The idea behind parameterization is to keep the structure of the change fixed, but make the names variable. Therefore, the need of modeling new architectural changes for repeated evolutions, evolutions using the same change structure with different names, is eliminated. For example, adding an interface to the connector *Scenario Control Interface* is a common evolution. However, we need to model a new architectural change for each new interface, similar to the one illustrated in Figure 2, as the names of the new interfaces would differ. With parameterization, we can make the description of the added interface a parameter and store the architectural change in the repository as a **template**.

A parameterized name/description starts with "@" symbol. Thus, we can make the architectural change of Figure 2 a template by changing the name of the added interface from *FirePreEmpt* to *@outsideControlEvent*.

4 Automated Steps

In the automated steps the evolution alternatives are generated, which starts with the designer specifying the type of the desired evolutions. These are input to the CDE which, then, fetches all architectural changes whose type matches to the specified type and prepares the graph system.

Binding Changes to Architecture: After locating the architectural changes matching the desired evolutions, CDE makes a list of parameters these take. Each parameter is presented to the designer in the following format: *@<parameter Name> _ <archtecturalChangeName> = .* At the RHS of the assignment, the designer fills the actual descriptions/names of the architectural elements. In case an architectural change is used multiple times, a unique instance name for each usage should also be specified at the right-hand side of the assignment.

Once the list is filled, *CDE* uses it to **bind** the architectural changes to the current architecture by substituting the parameters with the values specified at the RHS of the assignment. For example, with the specified value *@outsideControlEvent_addNewOutsideEvent = FirePreEmpt*, CDE replaces the parameter *@outsideControlEvent* with the value *FirePreEmpt* in the architectural change *addNewOutsideEvent*.

Generation of the Alternatives: After the binding, CDE forms the graph system containing the architectural changes as graph transformation rules and the architectural model as a graph. These are loaded to GROOVE which automatically generates the state-space, which has an initial state and final states. The initial state in our case is the input architectural model. The final states are states where none of the transformation rules match and, as such, they do not have any outgoing edges. These states contain the evolved architectural models; each final state is an **evolution alternative**. The path from the initial state to a final state, shows the applied architectural changes to reach that final state. Hence, we term such paths **evolution paths**.

Figure 3 shows the state-space generated from the architectural changes *AddFirePreEmpt* (Figure 2) and *addInterfacePreEmptHandler*. The later architectural change adds the interface *AddFireEmpt* and a new component *FireEventHandler* that only receives messages from this interface. This state-space contains two evolution alternatives, states *s2* and *s3*. We can see that state *s2* results from the application of the architectural change *addInterfacePreEmptHandler*. To reach the state *s3*, on the hand, two architectural changes are applied: the change *AddEventFirePreEmpt* and the ender change *AddReceiverFirePreEmpt*.

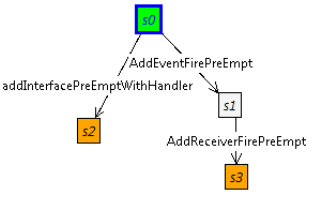

Fig. 3. The evolution paths for adding an interface

Querying/Scoring Alternatives: When the alternatives are generated, it is important to find the ones that have the desired structure and/or generated through a desired evolution path. For this, we have developed a querying mechanism, where the designers can query and give scores to the desired structure and/or architectural changes. Hence, the alternatives with the higher scores have a higher chance of being selected by the optimizer.

Two types of queries are possible with our system. The first type allows the designers to express a structure that is searched in the generated evolution alternatives. These are expressed as Prolog queries; we have extended GROOVE with a Prolog interpreter (using GNU-Prolog Java [11]) and implemented 8 rules that allows the user express queries based of the architectural models. When expressing the queries only the name parameter should be filled and a query should start with the rule *evolutionAlternative*. For example, we want the evolution alternatives that has a single component handling the events *FirePreEmpt* to have a high score. With, the following query we can locate such alternatives:

```
evolutionAlternative(S),connector("Scenario Control Interface",S,T)
component("PreEmpt Handler",S,C),outInterface("FirePreEmpt",T,I),
inInterface("FirePreEmpt",C,I2),link(I,I2,S),score(S,1)
```

The query above finds and gives the score 1 to all the evolution alternatives, which have a connector named *Scenario Control Interface* and a component named *PreEmpt Handler* linked to each other through the *FirePreEmpt* interface.

The second type of queries allows the designer to express desired/contraints over the evolution paths. These are expressed using Computational Tree Logic (CTL) [12]. Informally, a CTL formula consists of atomic propositions that are ordered with logical and temporal operators. In GROOVE, these automatic propositions consists of the labels of the transitions in the state-space; that is, the names of the applied architectural changes. We extended GROOVE's CTL evaluator to score the evolution alternative that occur after finding states that follow the formula. For example, the evolution alternatives generated from the evolution path where first the architectural change *addInterfacePreEmptHandler* followed by the architectural change *AddEventFirePreEmpt*, should get a low score because these contain the same interface added twice. We can query the evolution alternatives generated from such a path with the following formula:

(addInterfacePreEmpt ∧ (EF(AddEventFirePreEmpt)))
Score(-1)

Here, E means there exists at least one path and F means finally. The proposition *Score* is only used for expressing the given score and has no effect on the CTL formula. The path score is treated differently than the structural scores and by default both scores for all evolution alternatives are set to 0.

Calculation of Quality Attributes and Optimization: The cost of evolution for an alternative is calculated by summing up the estimated cost values of the architectural changes in its evolution path. Initially the cost is set to 0 as an attribute of a special node in the graph representation. Each architectural change is associated with a cost value. During the generation of the evolution path, the corresponding cost value is added by transformation rules for each applied change.

To estimate the reliability of a particular design alternative, each component is annotated with a reliability value R, where $0 < R < 1$. Based on these annotations and the connections among the components, reliability values are propagated through the interfaces. The reliability of a component is derived by multiplying its reliability value with the reliability values of other components on which it depends. The reliability measure for the overall system can be considered as a combined reliability of the components that interact with the user.

The selection of an evolution path considering multiple quality attributes and criteria (in this case, reliability, cost, structural query score and path query score) requires us to solve a multi-criteria optimization problem. The optimizer tool currently employs single-objective optimization techniques. As such, the objective function is specified as: maximize reliability such that the cost, structural query score and path query score are below/above certain thresholds.

5 Application of the Approach

In this section, we will consider two common evolutions to CMS and show how our approach helps the designers in selecting an evolution alternative. The first evolution is the addition of the scenario *Presidential Emergency Scenario* which requires an extra pre-emption event with a new crisis manager component. We

designed 5 architectural changes that can be used to add a new crisis manager with an event. These changes are not specific to the evolution of the presidential emergency scenario but are rather alternative ways of adding a new crisis manager and an event. The second evolution is the addition a resource allocation strategy based on the location of the resources which requires a new resource allocation component and a new resource location update event. We designed 6 architectural changes that can be used to add a new event and a new resource allocation strategy. We have checked in all these to the repository, then followed the automated steps to generate the evolution alternatives.

Generating the evolution alternatives: We checked out 11 architectural changes from the repository. We used the "full" CMS architectural model, which includes 20 scenarios and 5 resource allocation strategies; the graph model generated from this model contains 647 elements. With the 11 architectural changes, 23 evolution alternatives that can be reached by 41 evolution paths are generated by GROOVE (in 21 seconds with a dual core 2.3Ghz laptop). The number of evolution paths are greater than the number of evolution alternatives because some paths lead to the same alternative; GROOVE can detect and merge isomorphic states.

Pruning the alternatives: We want both the pre-emption and the resource location update events to be sent from the *outside* to the CMS. In addition to this, we prefer alternatives that use new components to propagate/handle these events as these alternatives are more modifiable (pre-emption is separated from scenarios). As the alternatives that have new components to handle these events are more modifiable, we give such alternatives the score 2 with the following query:

evolutionAlternative(S), connector("Scenario Control Interface", S, T), outInterface("FirePreEmpt", T, I1), outInterface("ResLocUpdate", T, I2), component("Resource Event Handler", S, C), inInterface("FirePreEmpt", C, IIn1), inInterface("ResLocUpdate", C, IIn2), link(I, IIn1, S), link(I2, IIn2, S), score(S, 2)

The architectural change that adds a new component to propagate the resource location update should be followed by the change that links this new component with the the connector *Resource Manager*. We can ensure that paths do not follow this constraint to get a low score with the following CTL formula:

(AddAllocationStrategyHandlerComponent ∧ !(EF(AddAllocationStrategyHandlerConnection)))

Score(-1)

Optimization and the selected alternative: Figure 4 presents an excerpt from the alternative selected by the optimizer, whose goal was to find the alternative with the lowest cost and the highest reliability, structure and path scores. Here, the resource location update and pre-emption events are send from the interfaces *ResLocUpdate* and *FirePreEmpt*.

This alternative has the highest reliability because crisis handling and resource updates are separated.It has the second highest cost; however, it has the highest structural (because, score 2 is given to the alternatives that use a differ-

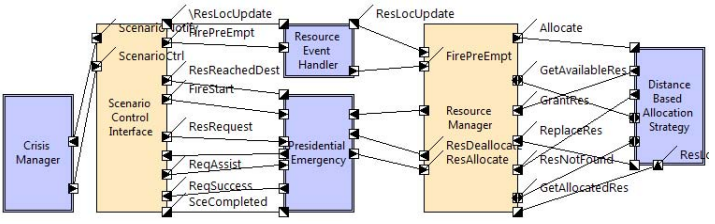

Fig. 4. The evolution alternative that uses the component *Resource Manager* to propagate the events about resources

ent component to propagate the events) and path (because, it does not violate the specified constraint) scores.

Discussion: In this application, 2 criteria were considered to evaluate 23 alternatives and 41 paths. The number of criteria and alternatives can be much higher for other systems. As such, manual generation and evaluation of these alternatives can be overwhelming. Moreover, it can be hard to differentiate among the isomorphic alternatives and reduce the design space manually as the number of alternatives increase.

6 Conclusion

We have introduced an approach for fostering reuse and automation in software architecture evolution. We have formalized architectural changes as graph transformation rules, which can be automatically applied on a graph representation of a software architecture. Thereby, our tools can generate possible evolution paths. Our toolset converts architecture descriptions and architectural changes specified in xADL to the corresponding graph representations and graph transformation rules, respectively. As such, the underlying theory is completely transparent to the designer. In addition to this, the evolution alternatives and paths can be queried and the alternatives that follow these queries can be scored. These scores and the quality attributes, such as cost and reliability are used by an optimizer to select the best evolution alternative.

References

1. Lehman, M., et al.: Metrics and laws of software evolution. In: METRICS, pp. 20–32 (1997)
2. Garlan, D., et al.: Evolution styles: Foundations and tool support for software architecture evolution. In: WICSA, pp. 131–140 (2009)
3. Kazman, R., et al.: The architecture tradeoff analysis method. In: ICECCS (1998)
4. Grunske, L.: Formalizing architectural refactorings as graph transformation systems. In: SNPD, pp. 324–329 (2005)
5. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. 44, 133–155 (2002)
6. Dashofy, E., van der Hoek, A., Taylor, R.: A highly-extensible, xml-based architecture description language. In: WICSA, p. 103 (2001)

7. Dashofy, E., et al.: Archstudio 4: An architecture-based meta-modeling environment. In: ICSE, pp. 67–68 (2007)
8. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis management systems: A case study for aspect-oriented modeling. In: Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on Aspect-Oriented Software Development VII. LNCS, vol. 6210, pp. 1–22. Springer, Heidelberg (2010)
9. Ciraci, S., van den Broek, P., Aksit, M.: Framework for computer-aided evolution of object-oriented designs. In: COMPSAC, pp. 757–764 (2008)
10. Grassi, V., Mirandola, R., Sabetta, A.: An XML-based language to support performance and reliability modeling and analysis in software architectures. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA 2005 and SOQUA 2005. LNCS, vol. 3712, pp. 71–87. Springer, Heidelberg (2005)
11. Gnu prolog java, `http://www.gnu.org/software/gnuprologjava/`
12. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)