Connector-driven gradual and dynamic software assembly evolution

Huaxi (Yulin) Zhang, Christelle Urtado, Sylvain Vauttier
LGI2P / Ecole des Mines d'Alès
Parc scientifique G. Besse - F30035 Nîmes cedex - France
{Huaxi.Zhang, Christelle.Urtado, Sylvain.Vauttier}@site-eerie.ema.fr

Abstract

Complex and long-lived software need to be upgraded at runtime. Replacing a software component with a newer version is the basic evolution operation that has to be supported. It is error-prone as it is difficult to guarantee the preservation of functionalities and quality. Few existing work on ADLs fully support a component replacement process from its description to its test and validation. The main idea of this work is to have software architecture evolution dynamically driven by connectors (the software glue between components). It proposes a connector model which autonomically handle the reconfiguration of connections in architectures in order to support the versioning of components in a gradual, transparent and testable manner. Hence, the system has the choice to commit the evolution after a successful test phase of the software or rollback to the previous state.

1 Introduction

The role of software architectures in software engineering becomes increasingly important and widespread [4]. Software architectures model the structure and behavior of systems. Architecture description languages (ADLs) [7] are not solely used during the design steps of the development process anymore, but also at runtime after deployment, to manage the evolution of application architectures. However, most ADLs do not fully support the evolution process of software systems. Software evolution is the process to change a software system from some version to a newer version. It consists in adding, deleting and replacing software components or connectors. Replacing a software component with one of its newer versions is the basic evolution operation that has to be supported. It is also the simplest and the more frequently used to improve software functionality, performance or reliability. However, component upgrading is an error-prone operation as it is difficult to guarantee (at least) the preservation of the functionality and quality. Some validation process should secure com-

ponent upgrading to avoid errors or regressions to be introduced by the new component version new version. Evolution mechanisms are provided in the main representative ADLs [8, 15, 16] to deal with component replacement in software architectures. However, existing works only propose limited support for evolution control that increases the chances for successful dynamic evolution. This paper considers applying an autonomic approach to manage and control evolution at runtime. A connector model is proposed which embeds the intelligence necessary to monitor and drive architecture modifications: we talk about connector-driven architecture evolution. Connectors are internally designed as expendable and reconfigurable component assemblies, able to manage various evolution concerns, depending on the requirements. A first application of these connectors is presented here: gradual evolution of assemblies in which new component versions are transparently tested before they effectively replace older ones.

The remaining of this paper is organized as follows. Section 2 defines the context of this research and spans existing related work. Section 3 specifies our objectives and describes the illustrative example we use throughout the paper. Section 4 describes our intelligent connector model and shows how it is component version evolution. Section 5 details our connector-driven gradual evolution process. Section 6 briefly describes the implementation of our model as an extension of the Julia implementation of the Fractal component model while Sect. 7 concludes with future work directions.

2 Context and related works

To develop a connector-driven gradual and dynamic evolution process requires an understanding of existing ADLs that support dynamic evolution and connector modelings.

2.1 Software evolution: definitions

Lehman [5] defines *software evolution* as the collection of all programming activities intended to generate a new

```
architecture SimpleExample is          system SimpleExample_1 is
 conceptual_components                   architecture SimpleExample with
   BikeGUI; Session;                        BikeGUI instance Basket1;
 connectors                                 Session instance Vendor;
   connector login is message_filter no_filtering;   end SimpleExample_1;
 architectual_topology
   connector login connections
     top_ports BikeGUI;
     bottom_ports Session;
end SimpleExample;
```

Figure 1. Sample architecture (left) and a conforming configuration (right) described using C2

version of some software from an older operational version. If these activities can be performed at runtime without the need for system recompilation or restart, evolution is called *dynamic software evolution*.

An explicit architectural view can enhance flexibility of software evolution. The *software architecture* of a program or computing system defines the structure of the system, which comprise software component types and connector types, the externally visible properties of those component types and connector types and the relationships among them [1]. Architectural *configurations* are connected graphs of concrete components (instanciated component types) and concrete connectors (instanciated connector types) that conform to the architectural structure [9]. Figure 1 provides an example of both an architecture and a conforming configuration using the syntax of the C2 ADL [10].

Existing work propose various typologies for evolution [2, 6], some of which we are going to use to precise which evolution is supportedin this paper . Dynamic software evolution may affect component or connector types at the architecture level or component or connector instances at the configuration level [9]. More precisely, component instance evolution potentially affects four elements of components: their name, interface, behavior protocols or implementation (as described by Palsberg and Schwartzbach [13]). Sometimes several of these elements can be simultaneously affected during evolution. Thus, rules that constrain valid evolution transitions migth concern several elements. For example, behavioral conformance allows any class to be a subtype of another if it preserves the interface and behavior protocol of all methods defined in the supertype. Lientz and Swanson [6] provide an orthogonal classification of evolution based on the reason that motivates the evolution. They identify three purpose-oriented evolution categories:

- corrective evolution, where the new version corrects errors/bugs identified in the old one,

- perfective evolution where the code of a component is evolved to improve non-functional attributes of the software (such as time performance),
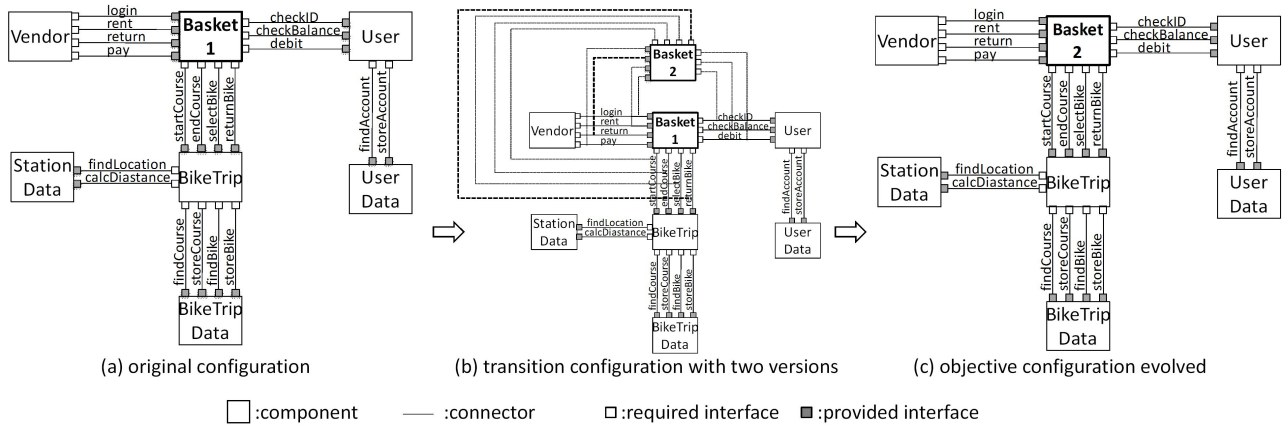
- adaptive evolution where components are adapted (or extended) to meet changes in their environment or in requirements on the software.

2.2 Dynamic evolution in existing ADLs

Few ADLs focus on supporting dynamic software evolution. Three representative approaches specify dynamic evolution in an ADL by providing a dedicated mechanism: decomposition and composition in ArchWare [11], an Architecture Modification Language (AML) in C2 [12] or the use of multi-version connectors in MAE [15]. ArchWare uses hyper code to examine the coherence of new configuration and implements evolution by decomposing the targeted configuration and then composing the new configuration. However, it focuses on configuration evolution and do not provide support for component substitution. In C2, an AML is proposed to manage configuration evolutions and to check their consistency. Component substitutions are controlled thanks to a subtype relationship [13] based on component specifications. However, it lacks to test the actual implementation of the new component. Cook and Dage [3] propose arbiters to run conjointly multiple versions of components that are transparently perceived by the rest of the system as single components. This approach is also proposed by MAE but uses connectors instead of arbiters. However, these two latter approaches aims at keeping old versions along with newer ones in the system for downward compatibility. The system thus grows in complexity and size. This is not suitable for a long-term, scalable solution, in which newer versions aim at replacing older ones after an interim test period.

The above representative ADLs contain useful insights for evolution, but they fail to account for several pertinent issues. There is: (1) no gradual process to ensure safe dynamic evolution, (2) no test of the actual implementation of the new component (functions might not be able to execute properly), (3) no mechanism to deal with system recovery when evolution fails.

Connectors bind components together and act as media-

(a) original configuration   (b) transition configuration with two versions   (c) objective configuration evolved

☐ :component   —— :connector   ☐ :required interface   ■ :provided interface

Figure 2. Example of basket evolution configuration comparison

tors [17]. As they clearly separate concerns of component computation and inter-component communication, they can be designed to meet non-functional requirement of system, such as evolution by reconfiguring the connection between components to realize their adding, deleting and replacing.

3 Goals and illustrative example

The example of a bike rental system illustrates the concepts of our evolution process throughout the paper (*cf.* Fig. 2). In this example, the *Vendor* component manages the user interface. It cooperates with the *Basket* component which handles user commands. The *Basket* component cooperates with the *User*, *BikeTrip* and *Account* components respectively to identify the user, check the balance of its account, assign an available bike and finally calculate the price of the trip when the bike is returned. Component *BikeTrip* manages the courses for each bike and each user and is designed to retrieve the location and calculate the distance between departure and arrival point.

The evolution objective is to replace the version 1 of the *Basket* component by its version 2. It implies changing the architecture from the original architecture represented on Fig. 2(a) to the target architecture represented on Fig. 2(c). Evolution concerns the *return* provided interface and its purpose is corrective. Contrary to most evolution approaches, we propose to deploy a transition architecture (*cf.* Fig. 2(b)) to smoothly switch from an architecture to another through a test period.

4 Intelligent connector model

In this section, our definition of the conceptual models of component architectures and connectors are presented. In our models, a software configuration is defined as a col-

lection of components and connectors which are contained in and managed by a container. These architectural entities are composed of *core* and *extension* elements. Core elements consist of classes which describe the basic, invariable part of the different kinds of architectural entities, i.e. their functional concerns. Extension elements are added to architectural entities in order to support the non-functional concerns which apply specifically, depending on the management policies defined by ADL descriptors, regarding for instance the dynamic evolution of components. Extension elements consist of meta-objects, *Controllers* and *Control interfaces*, that are added to the definition of components to modify and control their behavior. Figure 3 shows the class model of architectural elements.
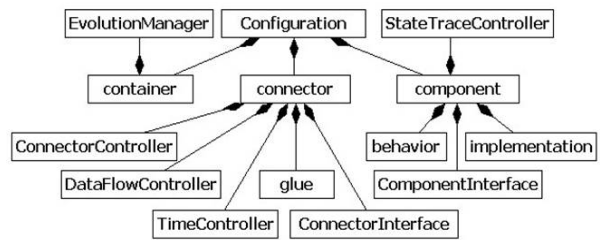
Figure 3. Class model of intelligent connector

A *container* is a core architectural element that encapsulates a set of *connectors* and *components* which describe a system as an assembly. It is extended by an *evolution manager* which manages the evolution of the its inner components and evaluates the feedback from its inner connectors. A *Component* has a name, a set of interface elements, an associated behavior protocol and an implementation as its minimal core requirements. Components should not communicate directly by referencing each other. Instead, they should use connectors. This minimizes coupling between

components and enables binding decisions to change without requiring component modification. Components are extended by *StateTrace controllers*. A StateTrace controller keeps track of the component states each time the component is modified. Its purpose is to recover from a failure and put back the component into a sound state to prevent system breakdown.

Connectors govern communication between components. Its core is composed by *interfaces*, *glues* and *properties*. Interfaces can be of two types: required or provided. Glues define the protocols of connection and interaction between connectors and components. Properties are non-functional attributes of connectors: they record version and type information. Connectors are extended in order to manage and control the evolution of each connected component, control the dataflow between components and collect test samples of component inputs and outputs. The extension of connectors consists of three controllers: the *connector controller*, the *dataflow controller* and the *time controller*. The connector controller controls evolution procedures for the connected components. The dataflow controller controls the dataflow that traverses the connector and records messages from each connected component to compare them in case evolution is declared to be corrective. The time controller records input and output time of each message that traverses the connector to compare them in case evolution is declared to be perfective.

These controllers are structured into three levels. *Evolution manager* is a top-level controller that controls the entire configuration evolution process by managing connectors and components. *Connector controllers* and *stateTrace controllers* separately control local evolution actions of respectively on connectors and components. *Dataflow* and *time controllers* are third level controllers that are used by *Connector controllers* to control and collect data on connectors.

5 Protocols of evolution

The main idea exposed in this paper is to get the original configuration evolve to a target configuration through a transition step. The transition configuration aims to test new component versions and either validate the evolution or invalidate it to return to the original state. To achieve this, we propose a four step evolution process controlled by an evolution manager:

1. *Preparation stage* to collect evolution description and deploy new component versions (connecting new component versions).

2. *Test stage* to collect observations from original and new component versions and determine if the new ver-

sions meet the requirements induced by the declared purpose of evolution.

3. *Observation stage* to maintain original component versions as backup in case new versions cause failures.

4. *Abandonment stage* to enact changes (disconnect the now useless original versions).

Each stage obeys two protocols: evolution and validation protocols. Evolution protocols define how connectors realize each evolution stage. Validation protocols are designed to examine the validation of activities in each stage.

5.1 Preparation Stage

Preparation is a vital aspect of the evolution process. Its objective is to change the original configuration into the transition configuration by collecting and deploying evolution descriptions.

Evolution description must define what to evolve and how. What to evolve refers to (1) references of old and new component versions, (2) interfaces changed and their change purpose (corrective or perfective), and (3) change test condition for corrective evolution. How to evolve refers the number of collected samples that are to be used during tests. In the example of Fig. 2, the *return* interface of *Basket* is modified in the new version for corrective evolution purposes: if type of bike is tandem then price is twice the normal price.

Secondly, the *evolution manager* deploys evolution descriptions according two connector groups: *changed group* and *unchanged group*. Changed group contains connectors which connect to changed interfaces like *rent*, *return* and *selectBike*. These connectors drive the evolution process. Unchanged group contains all other connectors like *login*, *pay*, etc.

At the validation step, the evolution manager collects feedback from all connectors. If all connectors connect successfully with new component version, the evolution process passes to next stage.

5.2 Test Stage

The test stage aims at examining the behavior of component new versions according to their evolution purposes. Test stage thus involves collecting observations on component new versions in connectors and evaluating these observation in the evolution manager. Measurements are decomposed into to two sub-stages: offline and online test.

Offline test is focused on the functioning of the new version, keeping it transparent to the rest of the system. Old version is *dominant*, which means that connectors just propagate old version's results to the rest of system, while new

Table 1. Offline and online data collected by controllers.

Controllers	Provided interface evolution	Required interface evolution
Dataflow Controller (DC)	(method, input, outputOld, outputNew)	(method, inputOld, inputNew, output)
Time Controller (TC)	(method, inputTimeOld, outputTimeOld, inputTimeNew, outputTimeNew)	

version is insulated from the system. *Online test* aims at adapting the system to new versions. As this entails risks of system failures, a state-trace mode is activated. All components' StateTrace controllers record the state modification of each component. When critical errors happen, the system rolls back to its previous safe state, as every component rolls back to its previous safe state.

Each test of a changed interface is controlled by the connector which connects with this interface. This includes dataflow control and data collection. Thus, according to the direction of interface (provided or required), test and evaluation behave differently. For instance (see Fig. 4), the *Basket rent* (P1) interface illustrates provided interface evolution, and its *returnBike* (R1) interface illustrates required interface evolution.

Provided interface evolution has either a corrective or per-
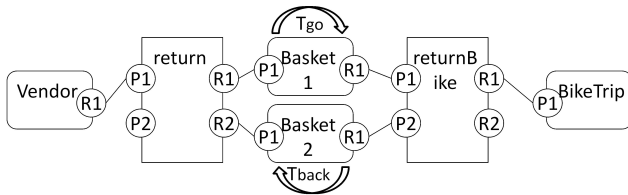
Figure 4. Net response time

fective purpose. The *return* connector (*cf.* Fig. 4) collects test data and controls the evolution of the *return* interface of the *Basket* component. During offline test, component *Basket1* is dominant. The dataflow controller of the *return* connector distributes incoming calls to two versions, while all connectors block outgoing calls from *Basket2* to keep it transparent to system. During online test, the situation is reversed: *Basket2* becomes dominant. The collected data elements are listed in Table 1.

Required interface evolution has a corrective purpose. The *returnBike* connector in Fig. 4 collects test data and controls the evolution of the *returnBike* interface of the *Basket* component. During offline test, the dataflow controller of the *returnBike* connector blocks *Basket2* incomings and let pass *Basket1* incomings. The outgoings of the *BikeTrip* component are sent to the two versions of *Basket*. During online test, the dataflow controller reverses the situation to make the new version (*Basket2*) become the dominant.

To evaluate a new version, observations encompasses various data, ranging from dataflow (corrective evolution) to execution times (perfective evolution) as listed in Table 1.

The dataflow controller is in charge of collecting input and output data. The time controller is responsible for collecting time for each input and output.

The evaluation of observations is handled by the evolution manager which collects all the observations from connectors. For corrective evolution, the new version must meets its predefined evolution condition. For perfective evolution, the new version must execute faster than the old version. Performance is calculated upon the net response times of components, $T_{response} = T_{go} + T_{back}$ (see Fig. 4). This means that the execution times of the functions called on other components is deducted from the observable response time of the component. After the new version passes the offline and online evaluation, the evolution process proceeds to the next stage.

5.3 Observation Stage

Observation stage is another feature used to further secure evolution. In this stage, the old version remains in the system, but only as a backup (it is not active anymore). The system is still in a state-trace mode. If a failure arises, the old version will be activated to replace the new version to roll back to the system to its previous sound state.

5.4 Abandonment Stage

Finally, the *evolution manager* either commits or abandons evolution. The unused version is disconnected and uninstalled.

6 Implementation in the Fractal component model

Our model is implemented as an extension of Julia, an open-source java implementation of the Fractal component model[1]. Fractal components are managed by controllers contained in the membrane of components. It thus was straightforward to adapt our model to this implementation. First, we added an *element-type controller* to specialize Fractal components into three sub-types — components, connectors and container. Then, we added specific controllers (as described in Sect. 4) to the adequate sub-type. The evolution manager implements controls evolution

---

[1] http://www.objectweb.org

through two mains functions. The *connect* function automatically generates a suitable connector and uses it to connect given components. The *cmpEvol* function triggers component version evolution: its parameters define the old and new component versions, the set of interfaces that are impacted by the change and the purpose of the evolution. Figure 5 shows a code sample that uses these functions.

```
// connect aComp to cComp1
Fractal.getEvolutionManager(container).
        connect (aComp, "r", cComp1, "f");

// evolve component cComp1 to cComp2
// (corrective evolution)
Fractal.getEvolutionManager(container).
  cmpEvol (cComp1, "f",
          cComp2, "corrective",
          "if(m.equal("bike")) rent(m)=1;");
```

Figure 5. Using the evolution manager

7   Conclusion and Perspectives

This paper presents and illustrates our connector-driven gradual and dynamic software assembly evolution process. Few existing ADLs fully support component replacement evolution process from its description to its test and validation. These ADLs do not support either component substitution, or any test phase, or the complexity induced by multi-version components. The evolution process proposed in this paper is a gradual, testable, transparent, repairable and spans the whole evolution process. It relies on a connector model that embeds the necessary intelligence to monitor and drive architectural configuration modifications.

Perspectives for this work are numerous. First, we plan to use the same evolution process to fully manage configuration evolution: component addition or suppression could benefit from an interim configuration as we propose. We also plan to tackle the "architecture drift" [14] problem which is a crucial coherence problem after architectural configuration evolution.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[2] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September 2005.

[3] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *Proc. of the 21st Int. Conf. on Software Engineering*, pages 203–212, Los Angeles, California, May 1999.

[4] D. Garlan. Software architecture: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[5] M. M. Lehman and J. C. Fernandez-Ramil. Towards a theory of software evolution - and its practical impact. In *Proc. Int. Symposium on Principles of Software Evolution*, pages 2–11, 2000.

[6] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[7] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, 2007.

[8] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proc. of the 21st Int. Conf. on Software Engineering*, pages 44–53, Los Angeles, CA, USA, May 16-22 1999.

[9] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[10] N. Medvidovic, R. N. Taylor, and E. J. Whitehead. Formal modeling of software architectures at multiple levels of abstraction. In *Proc. of the California Software Symposium 1996*, pages 28–40, April 17, 1996.

[11] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cimpan, B. Warboys, and R. M. G. Bob Snowdon. Support for evolving software architectures in the archware adl. In *Proc. of the 4th Working IEEE/IFIP Conf. on Software Architecture*, pages 69–78, Oslo, Norway, June 2004.

[12] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proc. of the 20th Int. Conf. on Software Engineering*, pages 177–186, Kyoto, Japan, 1998.

[13] J. Palsberg and M. I. Schwartzbach. Three discussions on object-oriented typing. *SIGPLAN OOPS Mess.*, 3(2):31–38, 1992.

[14] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[15] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *Proc. of the 2001 Symposium on Software Reusability*, pages 11–18, Toronto, Ontario, Canada, 2001.

[16] R. Roshandel, A. V. D. Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae–a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 13(2):240–276, 2004.

[17] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Proc. of the 3rd Int. Conf. on Configurable Distributed Systems*, pages 2–10, Annapolis, Maryland, 1996.