Connector-driven process for the gradual evolution of component-based software

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}@ema.fr

Abstract

Large, complex and long-lived software need to be upgraded at runtime. Addition, removal and replacement of a software component are the elementary evolution operations that have to be supported. Yet, dynamic changes are error-prone as it is difficult to guarantee that the new system will still work and that all functionalities and quality are preserved. Few existing work on ADLs fully support component addition, removal and substitution processes from their description to their 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 embeds the necessary mechanisms to do so. These connectors autonomically handle the reconfiguration of connections in architectures in order to support the addition, removal and substitution 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 [1]. Software architectures model the structure and behavior of systems. Architecture description languages (ADLs) [2] 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 architecture. 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 to improve its functionality, performance or reliability. It consists in addition, removal and replacement software components or connectors. These operations are error-prone. Indeed, it is difficult to guarantee the preservation of the functionality and quality of the software. It is also possible that the new software version be unstable after the introduction of new components or the removal of existing components. Some validation process should secure component adding, removing and replacing to avoid errors or regressions to be introduced by new component versions or instability due to component removals. Evolution mechanisms are provided in the main representative ADLs [3], [4], [5] to deal with elementary evolution operation 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 necessary mechanisms 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. An 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 and in which components connected to those that are going to be removed are tested without the component to be removed before it is effectively.

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 autonomic connector model. 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 Section 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.

2.1. Software evolution: definitions and typologies

Lehman [6] defines *software evolution* as the collection of all programming activities intended to generate a new 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*.

```
architecture SimpleExample is
 conceptual_components
   BikeGUI; Session;
 connectors
   connector login is message_filter no_filtering;
 architectural_topology
   connector login connections
     top_ports BikeGUI;
     bottom_ports Session;
end SimpleExample;
```

```
system SimpleExample_1 is
 architecture SimpleExample with
   BikeGUI instance Basket1;
   Session instance Vendor;
end SimpleExample_1;
```

```
SimpleExample_1.AddComponent(User)
SimpleExample_1.AddConnector(checkID)
SimpleExample_1.Weld(Basket1, checkID)
SimpleExample_1.Weld(checkID, User)
```

Figure 1. Sample architecture (left), a conforming configuration (top right) and an evolution description (bottom right) described using C2 and C2 AML

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 [7]. Architectural *configurations* are connected graphs of concrete components (instantiated component types) and concrete connectors (instantiated connector types) that conform to the architectural structure [8]. Figure 1 provides an example of both an architecture and a conforming configuration using the syntax of the C2 ADL [9].

Existing work propose various typologies for evolution [10], [11], that we are going to use to precise which evolution is supported in the system defined in this paper.

Dynamic software evolution might affect component or connector types at the architecture level or component or connector instances at the configuration level [8]. Evolution might concern the software semantics or its structure. *Semantic evolution* does not change the structure of the software system. It simply consists in replacing one or more components by their newer versions. *Structure evolution* changes the structure of the configuration by adding or removing a component that adds or removes functionalities to the system. Before evolution, the system needs to check the consistency of the new configuration by comparing component behaviors. A component's behavior protocols describe the possible orderings of (incoming and outgoing) method calls on the component's interfaces [12].

Lientz and Swanson [11] provide an orthogonal classification of evolution based on the reason that motivates changes. 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.

This paper deals with dynamic software evolution and covers semantic evolution (through component version substitution) as well as structure evolution (through component addition or removal) and both corrective and perfective evolution.

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 together with the original ADL: decomposition and composition in ArchWare [13], an Architecture Modification Language (AML) in C2 [14] or the use of multi-version connectors in MAE [4]. 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. This approach introduces additional risks during the evolution processes that makes evolution more complicate to implement. Evolution in C2 uses AML to manage the configuration evolution with a consistence examination of new configuration, and for component substitution, it uses a type theory-PL subtyping relationship [15] to check the consistency of component specification. Figure 1 shows an example of how the evolution can be implemented in the C2 AML syntax. However, this approach lacks to test the actual implementation of the new component. Cook and Dage [16] propose to keep multiple versions of a component running and uses arbiters to present to the system the image of a single component. Furthermore, Rakic [4] introduces this approach into an ADL (MAE) and uses connectors instead of arbiters. This multi-version approach improves component substitution evolution by connecting multiple versions of components at the same time in the system. Both approaches limit the changes on new component versions: they constrain the outputs — the values returned by function executions — of new versions to remain unchanged. In these two latter approaches however, component versioning aims at keeping old versions along with newer ones in the system for downward compatibility. The system thus grows in complexity and size which is not

suitable in our case where newer versions aim at replacing older ones after an intermediate 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 real test of the actual new configuration to try and detect functional problems that might be introduced by a new component or instability that might result from some component suppression,
3) no use of evolution semantics to drive the evolution process
4) no mechanism to deal with system recovery when evolution fails.

2.3. Connector semantics

Connectors bind components together and act as mediators between them [17]. They clearly separate concerns of component computation and inter-component communication. Connector ends are typed by a set of interfaces (a set of connection points) that define the signatures of functionalities that the connector passes through. Each interface has a direction, either provided or required. A provided (*resp.* required) interface of a connector can connect to a required (*resp.* provided) interface of a component, if their types are identical (or, in a wider sense, compatible [18]). If the set of interfaces at one end of the connector is the same but has opposite directions as the set of interfaces at its other end, then the connector just lets functionality calls pass through. If not, the connector is said to do some functional adaptation. For simplicity's sake, in this paper, connectors are pass through connectors; they don't adapt function calls (yet).

Representative connectors models are UniCon [17], SOFA [19], Wright [20] and COSA [21]. Beyond the basic binding function of connectors, each of these models has its specificity. UniCon has a predefined taxonomy of connectors, either simple or complex (pipes, remote calls, etc.). SOFA has a composite connector model that allow connectors to be described by an inner component architecture. Wright models connector glue and event trace specification with CSP. COSA eases the building of numerous connector types by composing simple connectors. None of these models, however, provide system life-cycle related services.

As connectors clearly separate concerns of component computation and inter-component communication, we claim they can be designed to meet non-functional requirements of systems, such as adding security (encrypting messages) or life-cycle related services. Our work proposes a connector model which embeds a system evolution support service. It reconfigures the connections between components to add new component versions, remove components or replace components by one of their (newer) versions. This is why our evolution process is said to be connector-driven.

3. Goals and illustrative example

Dynamic software evolution requires that evolution be performed at runtime. This operation is risky as the new version might introduce errors. These errors might come from the functional problem of new configuration introduced by new components or new component versions, or from the inconsistency of new configuration caused by the removal of components. Despite this, most research on dynamic evolution mainly focuses on configuration consistency test at the architecture level, before the implementation of evolution.

To our knowledge, no approach proposes a gradual process based on the configuration level that supports dynamic evolution and include correction tests of both the new configuration and the newly introduced components; no ADL uses the semantics of evolution operations to describe, test and measure the quality of the evolved software; and no connector model embeds mechanisms to monitor and drive evolution.

The work presented in this paper aims at developing a connector driven gradual and dynamic evolution process which fully covers evolution, from its description to its test and validation. It focuses on three elementary change operations — component addition, removal and replacement — and tests to the validity of changes before committing them. As an initial application of our work, each evolution consists in a single change operation. When component replacement is concerned, the evolution semantics proposed by Lientz and Swanson [11] is used to measure the adequateness of the new component. Our system covers two of their evolution purposes: perfective evolution and corrective evolution. It uses this information on evolution semantics to test the resulting configuration.

Furthermore, our evolution process is required to have the following characteristics:

1) gradual: have the architecture evolve smoothly through a transitional phase,
2) testable: be able to test desired changes (component evolution) and validate them before they become effective,
3) transparent: have changes (introduce new component and remove existing component) during the transitional phase and test phase remain transparent to the software system,
4) self-repairable: prevent fatal failures using a system recovery mechanism. This implies that components have the capability of recording their internal states and recovering from this recorded state when needed.

In order to support such evolution process, connectors need to supply following functionalities:

1) connect and disconnect multiple component versions at runtime,
2) test new configuration and collect observation data,
3) control the dataflow to make component test transparent to the system.

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 course when the bike is returned. Component *BikeTrip* manages the courses for each bike and each user and is designed to retrieve location and calculate the distance between departure and arrival addresses.

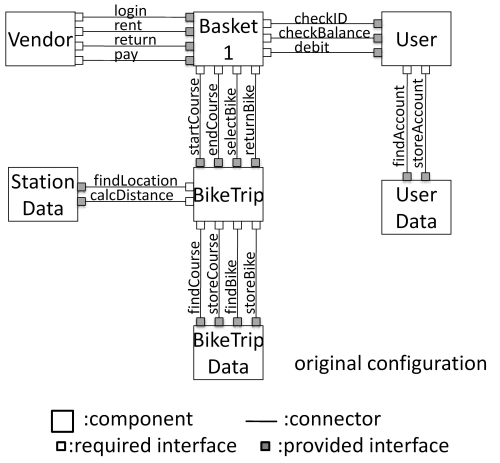

Figure 2. Example of a bike rental system configuration

4. Connectors that drive the evolution process

In this section, our definition of the conceptual model of component architectures and connectors is presented. In our model, a software configuration is defined as a collection 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.

A *container* can be seen as the component at the top of the composition hierarchy. It contains the description of the entire software system and thus contains a set of *connectors* and *components* which describe a system as a configuration. It is extended by an *evolution manager* which manages the evolution of its inner components and evaluates the feedback from its inner connectors.

A *component* has a name and a set of interface elements. There are two kinds of components: *primitive* and *composite*. A primitive component is a leaf component that is directly implemented. A composite component is built from the interconnecting o more primitive components and connectors as a substructure. Primitive components are extended by *StateTrace controllers*. *StateTrace controllers* keep track of the successive states of primitive components each time they are modified. They allow to recover from a failure and put back the components into a sound state to prevent system breakdown.

Connectors govern communication between components. A connection is the binding between provided and required interfaces of components and connectors. Indeed, components should not communicate directly by referencing each other. Instead, they should use connectors which minimize coupling between components and enable binding decisions to change without requiring component modification. The interfaces of two component which are connected by a connector need to be compatible: the names of methods should be the same. The connection should be chained as is: a component's required interface connects to a connector's provided interface and the required interface of the connector connects with another component's provided interface.

Connectors are extended 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 of 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 its 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 connectors and components. *Dataflow* and *time controllers* are third level controllers that are used by *Connector controllers* to control and collect data on

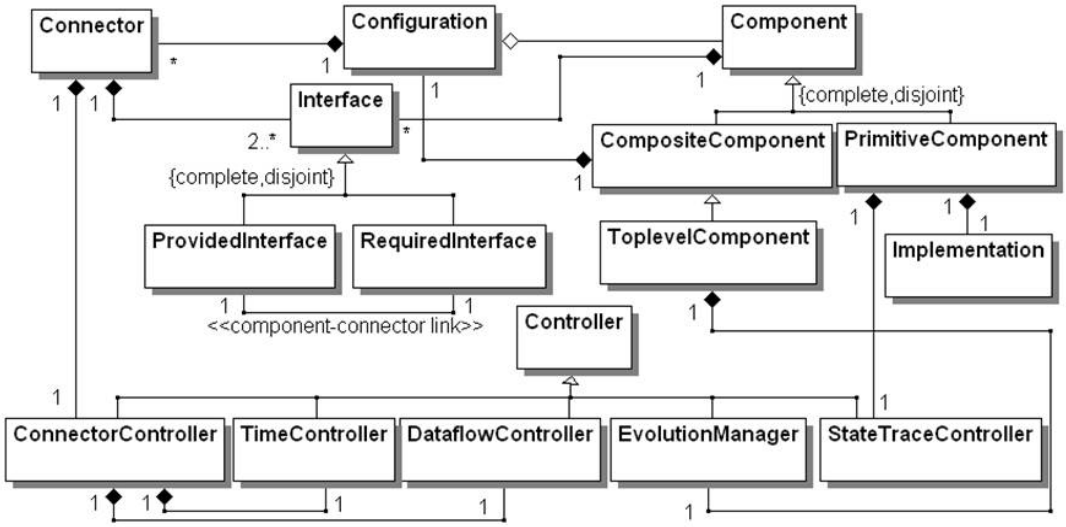

Figure 3. Class model of connectors

connectors.

5. Protocols of the evolution operations

One of the main ideas exposed in this paper is to have the original configuration evolved into an objective configuration through a transition step. The transition configuration aims to test new component versions and either validate the evolution (commit changes) or invalidate it to rollback to the original state.

To achieve this, we propose a four step evolution process controlled by an evolution manager, depicted in Fig. 4. The *preparation stage* collects evolution description and builds the transition configuration. The *test stage* collects observations from newly introduced components and components affected by changes (those connected to changing components) to determine if new components work correctly (in case of component addition), new component versions meet the requirements induced by the declared evolution purpose (in case of component substitution) or the new configuration is stable (in case of component removal). The *observation stage* maintains original component versions as a backup in case new versions cause failure. The *commit stage* enacts changes (disconnect the now useless original versions).

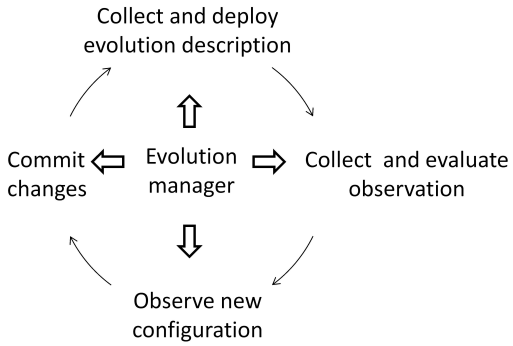

Figure 4. Life cycle of evolution actions

Each stage obeys two phases: evolution and validation. The evolution phase defines how connectors drive the evolution process during each stage. The validation phase determines if the activities performed during each stage are valid or not.

5.1. Component substitution

Component version substitution potentially affects four elements of components: their name, interface, behavior protocols or implementation (as described by Palsberg and Schwartzbach [15]). In our system, component substitution amounts to replace an old version of a component by one of its newer versions in order to meet the declared evolution requirements (corrective evolution or perfective evolution). Thus, the test focuses on the two evolution motivations which affect the measures our evolution manager can perform to guarantee the feasibility of the thought evolution.

Example. The objective of this evolution is to replace version 1 of the *Basket* component by its version 2. It implies changing the architecture from the original architecture represented on Fig. 2 to the objective architecture represented on Fig. 5b. 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. 5b) to smoothly switch from an architecture to another through a test period.

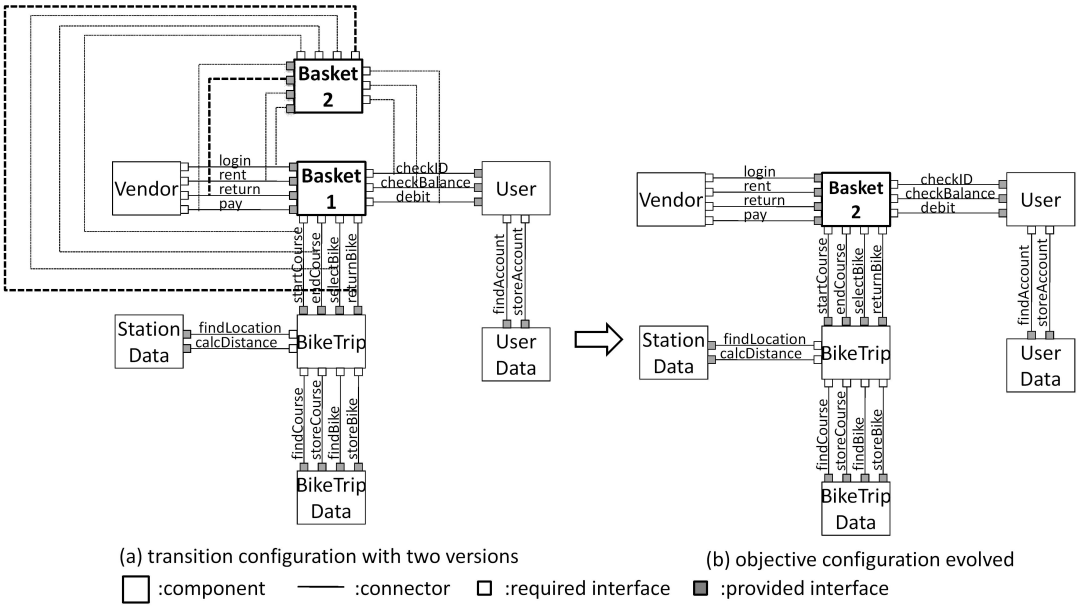

Figure 5. Example of component substitution evolution

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 description.

The first step is to collect evolution description. 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 changed to correct (corrective evolution) its protocol for *Basket* version 2 with the following condition: if type of bike is tandem then price is twice the normal price. The collected evolution description is shown in Table 1.

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.

Test stage. Test stage aims to examine the behavior of new component version according to its evolution purpose. Test stage thus involves collecting observations on new component versions in connectors and evaluating this observation in the evolution manager. The measurement of new component versions focuses on functioning and 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 system. Old version is *dominant*, which means that connectors just propagate old version's results to the rest of system, while new version is insulated from the system.

Online test aims at adapting the system to new version: new version becomes dominant. As this entails risks of system failure, a state-trace mode is activated. All components' StateTrace controllers record the state modification of each component during online test. When critical errors happen, the whole 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

Table 1. Evolution description of example

Components	**Old**	Basket version 1
	New	Basket version 2
Change interface	**Purpose**	**Condition**
selectBike		
rent	perfective	
return	corrective	if bike∈Tandem then Basket2.return(bike)= 2*Basket1.return(bike)

evaluation behave differently. We use a part of the example to explain how dataflow is controlled by connectors (Fig. 6). The *Basket rent* (P1) interface illustrates provided interface evolution, and its *returnBike* (R1) interface illustrates required interface evolution.

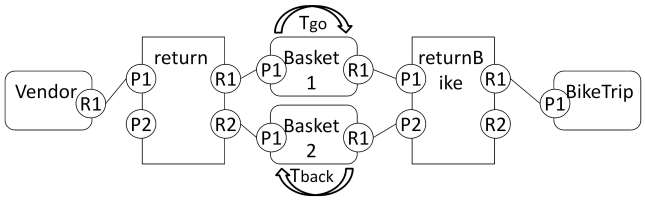

Figure 6. Net response time

Provided interface evolution means that a component's provided interface is evolved with either a corrective or a perfective purpose. The *return* connector (*cf.* Fig. 6) 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. For example, the *return* connector blocks the response of *Basket2* and the *returnBike* connector blocks the incoming calls of *Basket2*. During online test, the situation is reversed: *Basket2* becomes dominant. The collected data elements are listed in Table 2.

Required interface evolution means that a component's provided interface is evolved with a corrective purpose. The *returnBike* connector in Fig. 6 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 dominant.

To evaluate a new version, observations encompasses various data, ranging from dataflow (test for corrective evolution) to execution times (test for perfective evolution) as listed in Table 2. The dataflow controller is in charge of collecting data from incoming calls which comprises the calling method and inputs and outputs of old and new versions for this calls. The time controller is responsible for collecting time for each input and output. Furthermore, in order to obtain the needed number of observations, each data collection by connectors will continue until all connectors in the changed group have enough observation samples (as defined in the preparation stage).

The evaluation of observations is handled by the evolution manager which collects all observations from connectors and thus have a global view to precisely calculate the time spent in each function in order to test the behavior of the new version. 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. But the time of response measured is not really the response time of the evolved interface. For example, when the *return* interface of component *Basket1* is called by component *Vendor*, it will call the interface *returnBike* before it returns an answer to the *Vendor* component. Thus, the response time of the *return* interface includes the response time of the interface *returnBike*. In order to get net response times for the component, the formula to calculate the response time is $T_{response} = T_{go} + T_{back}$ in Fig. 6. After the new version passes the offline and online evaluation, the evolution process proceeds to next stage.

Observation and commit stage. Observation stage is another feature used to further secure evolution. In this stage, the old version still remains in the system, but only as a backup (it is not active anymore). The system is still in state-trace mode. If a critical error of new version arises and makes the system fail, the old version will be activated to replace the new version and the whole system will roll back to the previous sound state. Finally, the *evolution manager* either commits evolution after the above stages (if the new versions passes all stages) or abandons evolution and disconnects the new version (rollback). The unused version is disconnected and uninstalled.

5.2. Component addition

Component addition aims to add some functionality to the system. This change may cause system failure if there are errors in the new component functionality or if the new component is not compatible with its connected components. The evolution process mainly tests the two aspects: the functionality of the new component and the compatibility of the new component to its connected components.

Example. The evolution objective is to add the *Station Data* to the system. It implies changing the configuration from the original architecture represented on Fig.7a to the objective configuration Fig.7b.

Preparation stage. The preparation stage aims to both introduce a new component into the system and connect it. The addition of a component must be accompanied by the addition of new connectors which connect the new component with existing components. At first, the preparation actions for component addition are the same as in other ADLs :

1) Add the component to the system,
2) According to the connected interfaces, generate connectors,
3) Connect the new component and connectors and the existing components.

Table 2. 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)	

Then, in order to test the new component, we need to find its *connected components* and the *affected interfaces* from its connected components, the interfaces of the connected components that are indirectly involved in the behavior of the interfaces of the changed component. We uses the *affected connectors* which connect with affected interfaces to test new version's adaptiveness. The new generated connectors which connect new component with other components called *connected connectors*. Table 3 summarizes this data for the addition of the *Station Data* component of our example.

Table 3. Affected connectors and components when adding the Station Data component.

Connected connectors	findLocation, calcDistance
Connected components	BikeTrip
Affected connectors	selectBike, returnBike

Test stage. The test stage focuses examine the functionality (offline) and adaptiveness (online) of the new component.

- Offline test: test its functionality. Connected connectors authorize the input of new component but do not return the response to test whether the responses are produced correctly.
- Online test: test its adaptiveness with connected component. During this test stage, each affected connector sends twice the input to the connected component. At first, the controlling connectors block the connections between the new component and connected components. The affected connectors collect responses and send them to the system. In a second phase, controlling connectors authorize the connection between the new component and existing components. The affected connectors collect the response but do not send it to the system. Between these two phases, components record their states and after the first phase, return to their original states to pretend that the first phase did not exist. After test, affected connectors send all this data collection to the evolution manager which compares the two sets of responses and decides whether to effectively add the component or not.

Observation and commit stage. During this stage, the system remains in recovery alert state. All state controllers record the states of components to prevent errors from the new component. If the new component works well during the observation stage, the system will enter its actual working state.

5.3. Component removal

Component removal aims to suppress an unused component from the system. This change may leave the system incomplete and even prevent it from continuing to work. The evolution test evaluates if the new configuration works correctly without the component to remove.

Example. We reverse the example of the component addition. The evolution objective is to remove the *Station Data* from the system. It implies changing the configuration from the original architecture represented on Fig. 7b to the objective configuration Fig. 7a.

Preparation stage. The preparation stage is similar to component addition's. We need to find the connected connectors, connected components, and affected connectors.

Test stage. This test mainly focuses on the connected components to evaluate whether they can work correctly without the removed component. All affected connectors send twice the same input to connected components. At first, connected connectors authorize input calls from connected components. Affected connectors record this response and send it back to the system. In a second phase, connected connectors block the input calls from connected components. Affected connectors record the responses of connected components which do not successfully call the removed component. After test, affected connectors send all this data collection to the evolution manager which compares the two sets of responses and decides whether to effectively remove the component or not.

Observation and commit stage. The removed component will remain in the system but isolated by connected connectors, to prevent connected components from generating instability errors. If the system has no problem during the observation stage, the removed component and connected connectors will be effectively be removed from the system.

6. Implementation as a Fractal extension

Our model is implemented as an extension of Julia, an open-source java implementation of the Fractal component model[1]. In the Fractal component model, everything has to be a component. Fractal components are managed by controllers contained in the membrane of components. It thus was straightforward to adapt our model to the implementation. First, we kept all existing Fractal controllers.

1. `http://www.objectweb.org`

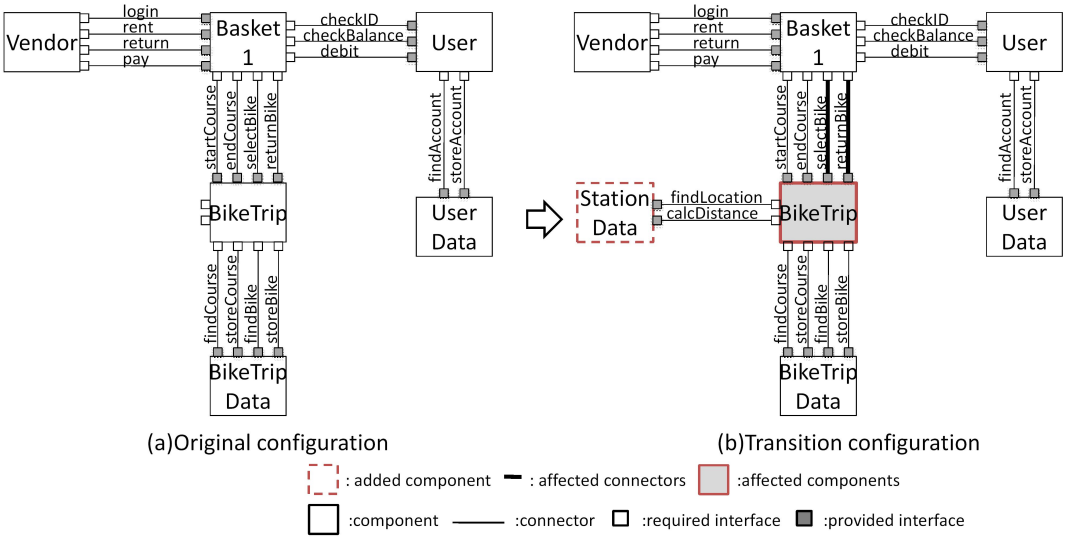

Figure 7. Example of component addition

Then, we added an *element-type controller* to specialize Fractal components into three sub-types — components, connectors and container — by storing the type information in this *element type controller*. Then, we added specific controllers (as described in Sect. 4) to the adequate sub-type.

```
public interface EvolutionControlInterface
{
void cmpEvol (Component oldC, String itfNames,
              Component newC, String ePurposes,
              String correctiveDescription);
void add (Component a, String [][] connections);
void remove (Component a);
}
```

Figure 8. Interface of the evolution manager

The evolution manager implements the *EvolutionControlInterface* to control evolution through two mains functions as presented in Fig. 8. The *cmpEvol* function triggers component version evolution: it is given as parameters the old and new component versions, the set of interfaces that are impacted by the change and the purpose of the evolution (corrective, perfective or both). The *add* function triggers component addition evolution: firstly, the new component will be added to system, secondly, it automatically generates a suitable connector and uses it to connect given components (connections presents which component's interface it should be connect). The *remove* function triggers component removal evolution.

Figure 9 shows a code sample that uses these functions.

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. Our contribution is twofold. Firstly, we propose a gradual, testable, transparent and repairable evolution process which fully supports evolution operations from the specification of evolution semantics to the test and validation of changes. Secondly, we introduce a connector model which embeds the necessary mechanisms to monitor and drive architectural configuration modifications. We use evolution semantics to automatically build a connector that measures, *in situ*, the safety of the desired changes and either commits or rollbacks them.

Perspectives for this work are numerous. We plan to use the same process to support complex evolution operations [22] that are composed of several successive ele-

```
// 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 9. Use of the evolution manager

mentary operations that are all semantically related. The semantics of these changes might also be exploited to drive the change process. We also plan to tackle the "architecture drift" [23] problem which is a crucial coherence problem after architectural configuration evolution. It will consist in synchronizing the architectural view of the software system to the actual evolved implementation, thus allowing to enrich the architectural description of the software from the changes that succeeded.

References

[1] D. Garlan, "Software architecture: a roadmap," in *The Future of Software Engineering*, A. Finkelstein, Ed. ACM, 2000.

[2] 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.

[3] 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'l Conf. on Software Engineering*, LA, CA, USA, May 1999, pp. 44–53.

[4] 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*, Toronto, Canada, 2001, pp. 11–18.

[5] 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.

[6] M. M. Lehman and J. C. Fernandez-Ramil, "Towards a theory of software evolution - and its practical impact," in *Proc. Int'l Symposium on Principles of Software Evolution*, Kanazawa, Japan, November 2000, pp. 2–11.

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

[8] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE TSE, 26(1):70–93*, 2000.

[9] 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*, april 1996, pp. 28–40.

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

[11] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Addison-Wesley, 1980.

[12] F. Plasil and S. Visnovsky, "Behavior protocols for software components," *IEEE TSE, 28(11):1056–1076*, 2002.

[13] 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 4^{th} Working Conf. on Software Architecture*, Oslo, Norway, June 2004, pp. 69–78.

[14] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proc. of the 20^{th} Int'l Conf. on Software Engineering*, Kyoto, Japan, 1998, pp. 177–186.

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

[16] J. E. Cook and J. A. Dage, "Highly reliable upgrading of components," in *Proc. of the 21^{st} Int'l Conf. on Software Engineering*, Los Angeles, CA, May 1999, pp. 203–212.

[17] M. Shaw, R. DeLine, and G. Zelesnik, "Abstractions and implementations for architectural connections," in *Proc. of the 3rd Int'l Conf. on Configurable Distributed Systems*, Annapolis, MD, 1996, pp. 2–10.

[18] G. Arévalo, N. Desnos, M. Huchard, C. Urtado, and S. Vauttier, "Precalculating component interface compatibility using FCA," in *Proc. of the 5^{th} int'l conf. on Concept Lattices and their Applications. CEUR Workshop Proc. Vol. 331*, J. Diatta, P. Eklund, and M. Liquière, Eds., Montpellier, France. 2007, October 2007, pp. 241–252.

[19] D. Bálek and F. Plášil, "Software connectors and their role in component deployment," in *Proc. of DAIS'01*. Krakow, Poland: Kluwer, September 2001, pp. 69–84.

[20] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. on Software Engineering and Methodology, 6(3):213–249*, 1997.

[21] M. Oussalah, A. Smeda, and T. Khammaci, "An explicit definition of connectors for component-based software architecture," in *Proc. of the 11th IEEE Int'l Conf. and Workshop on the Engineering of Computer-Based Systems*. Brno, Czech Republic: IEEE, May 2004, pp. 44–51.

[22] C. Urtado and C. Oussalah, "Complex entity versioning at two granularity levels," *Information Systems, 23(2/3):197–216*, 1998.

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