

Code Generation From Architectural Multi-views Description

Abdelaziz Gacemi, Département GIP, Ecole des Mines de Douai, France
Abdehak Seriai, Département GIP, Ecole des Mines de Douai, France
Mourad Chabane Oussalah, LINA, Université de Nantes, France

The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components. It typically plays a key role as a bridge between requirements and code. Practitioners have come to realize that having a good architectural design is a critical success factor for complex system development. To improve design at architectural level, we have proposed a description model based on the view notion. Through this model, it is possible to describe both components and connectors according to several views. In this paper, we deal with the code generation from architectural specification obtained via our multiviews description model.

1 INTRODUCTION

To describe the software architecture of complex software systems, formal and expressive notations are needed. Architectural description languages (ADLs) have been proposed as the answer. They provide a formal modeling notation for representing and analyzing architectural designs.

To improve architectural design, we have proposed in [2] a multiviews description model which aims to allow the separation of concerns at software architecture level. This separation is based on the view concept [6].

As any other modeling approach, the ultimate goal of an architectural design is to produce the implementation from architectural description. This mapping between the architecture and its implementation becomes more useful if the architecture description model supports advanced aspects of description like the separation of concerns in our model. In this paper, we deal with the code generation from architectural specification obtained from our multiviews description model.

The remainder of this paper is organized as follows. Section 2 presents some related work. Section 3 gives an overview of our multiviews description model. Section 4 demonstrates how to produce implementation-level description from architectural-level one. Conclusions round out this paper.

2 RELATED WORK

There are three areas of related work : Architectural description, separation of concerns, and code generation.

- *Architectural description languages (ADLs)* : In the past few years, ADLs have become an area of intense research in the software architecture. A number of ADLs have been proposed. Among the most representative ADLs, we can enumerate Unicon [8], Wright [1].
- *Separation of concerns (SOC)* : SOC is a concept that is at the core of software engineering. It refers to the ability to identify, encapsulate, and manipulate parts of software that are relevant to a particular concern [7]. The view concept is a very widespread form to reach the separation of concerns [6]. It appears in various forms. Among them, we can quote Subject Oriented Programming (SOP) [4] and Aspect Oriented Programming (AOP)[5].
- *Code generation from architectural-level specification* : There are two approaches. The first one consists of a direct code generation. The ADL Unicon [8] adopts this approach. Its compiler allows code generation from connector abstractions. However, only set of predefined connectors are supported. The second one consists of the use of an intermediate notation, like the object notation, which is close to implementation. In [3], Garlan and *al.* describe principal the strategies to map architectural description into the object modeling notation UML.

3 AN OVERVIEW OF THE MULTIVIEWS DESCRIPTION MODEL

The key concept of our description model [2] is the view concept which permits to address only concerns that are of interest, ignoring others that are unrelated. Our model defines a style of description organized in two stages. The first one consists of describing, in an *independent manner*, the various architectural element views¹. This yields several descriptions. Each description belongs to an architectural element view according to a given viewpoint. The second consists of describing the assembly of the resulting views defined in the first stage. Like Wright [1], our description model allows behavior description of the architectural elements. Thus, the assembly of the views consists of *coordinating* various views behaviors defined on a given architectural element. This task of coordination is ensured by a *coordinator*. So, the structure of multiviews description for components and connectors is defined by two kinds of sections: one to describe the views and another section to describe the coordinator.

¹We employ the term "architectural element" to indicate both components and connectors



Views description

In the case of a component, each view is defined, on the one hand, by a set of *ports* representing its interface according to a given viewpoint and, on the other hand, by a *computation* which provides a more complete description of what is done according to this viewpoint.

As for a connectors, there is a difference to take into account. Indeed, a connector has a view where it is seen as an architectural element modeling interactions between a set of components. This view is defined, on the one hand, by a set of *roles* representing its interface according to a given viewpoint. Each role defines the behavior of one participant in the interaction. On the other hand, the view is defined by a *glue* defining how the participants collaborate together to create an interaction. However, in other views, this same connector is seen as a component interacting with other components in order to carry out a given functionality. In this case, a view is defined by ports and a computation.

The specification of view behavior is based on process algebras to have a formal definition of the behavior. The behavior description of the entities constituting a view, namely computation, port, glue and role, is similar to Wright. These entities are first class concepts and are represented by processes. Indeed, a process is an behavior pattern. The basic unit to specify this behavior is *the event*. It could either be *initiated* by the process, in this case the name of the event will be written with overbar "*initiatedEvent*", or *observed*, therefore initiated by other processes, and in this case, the name of the event will be written without overbar "*observedEvent*".

Coordination description

In our approach, views are described in an independent manner. Nevertheless, it could exists between these views some dependencies which are not described. Those are, in our case, of behavioral nature, *i.e.*, dependencies related to the behavior evolution of different views. This is why, we proposed to express, explicitly, these dependencies in terms of coordination. The coordination description introduces, in our case, constraints which define a temporal order between the execution of the various views. The coordinator behavior is specified, in our model, in terms of a set of *coordination rules*. Each rule specifies the two following aspects:

1. the event name that is responsible for starting coordination. This event is observed by the coordinator on observable points which are, in our case, ports, roles, computations and glues.
2. coordination actions that the coordinator engages as soon as it observes this event.

Example

Our example consists of “synchronous procedural call” connector, noted *SPC* (see Figure 1). This connector allows a component to call a service provided by an another component while it is charged to ensure the synchronization of this call with other ones. One of applications of this type of connectors is to allow two components to *exclusively* interact with a shared resource.

```

Connector SPC{
  As ProcedureCallView :
    Role Caller =  $\overline{\text{call}} \rightarrow \text{return} \rightarrow \text{Caller}$ ;
    Role Definer =  $\text{call} \rightarrow \text{return} \rightarrow \text{Definer}$ ;
    Glue =  $\text{caller.call} \rightarrow \overline{\text{definer.call}} \rightarrow$ 
            $\text{definer.return} \rightarrow \overline{\text{caller.return}} \rightarrow \text{Glue}$ ;

  As SynchronisationView :
    Port Token =  $\text{acquire} \rightarrow \text{Token} \mid \text{release} \rightarrow \text{Token}$ ;
    Computation =  $\text{token.acquire} \rightarrow \text{token.release} \rightarrow \text{Computation}$ 

  Coordinator:

  On ProcedureCallView.caller.call Do
     $\overline{\text{SynchronisationView.computation.token.acquire}} \rightarrow$ 
     $\text{ProcedureCallView.glue.caller.call} \rightarrow \text{Coordinator}$ 

  On ProcedureCallView.glue.caller.return Do
     $\overline{\text{SynchronisationView.computation.token.release}} \rightarrow$ 
     $\text{ProcedureCallView.caller.return} \rightarrow \text{Coordinator}$ 
}

```

Figure 1: Communication connector with synchronization

The *SPC* connector can be described according to two views. The first one, noted “*ProcedureCallView*” describes a basic interaction protocol (a basic method call). According to this view, the connector has two roles, “*caller*” and “*definer*”. The second one, noted “*SynchronisationView*”, shows how this connector achieves the synchronization of the interaction. According to this view, this connector is seen as a component which has a port named “*Token*”. Via this port, the connector acquires a token if it is not already held by another connector and releases it afterward. Of course, the connector remains blocked if the token is not released.

As we have mentioned before, the description of the second view is completely independent of the first one. It is described according to a reasoning which is independent of that of the first view. It remains now to define coordination rules. For this case, two rules are necessary to describe the following behavior:

As soon as the event “call” appears on the role “Caller”, the coordinator redirects the flow of execution toward the view “SynchronisationView” in order to acquire a token. This procedure of the token acquisition involves interaction blocking as long



as the token is not acquired. Once the token is acquired, the coordinator starts again the protocol of procedural call. As soon as the coordinator intercepts the event “return” on the “glue”, which means that the protocol ended, the coordinator redirects the execution flow toward the view “SynchronisationView” in order to release the token. Once released, the coordinator emits the event “return” to the role “Caller” to return result to the component connected to this role.

4 CODE GENERATION FROM MULTIVIEWS DESCRIPTION

We now define rules to produce code from a given multiviews description. We present at first the mapping between views description and their implementations. Then, we show how coordination can be interpreted at the implementation-level.

Mapping of view description

Regarding structural aspects of description, we defined general rules to achieve the mapping. Indeed, all design elements that are first-class entities at architectural-level are also first-class entities at implementations-level. So, ports, roles, computation and glue are mapped into classes.

In addition, a view is mapped into a composite class in order to assure that concepts which form this view will share the same identity at implementation. The table 1 illustrates code that corresponds to the two views of the connector SPC, described above.

As for behavioral aspects of description, the mapping strongly depends on the behavior description semantic. As illustrates table 1, some events are mapped into class method, e.g the event “call”. Others are mapped into a line code, e.g “return”. However, the general behavior pattern for each element design is correctly mapped.

Mapping of coordination description

As we noted above, coordination is an abstract composition description. It does not precise which mechanisms are used to compose views but it describes how to compose views. Then, the coordination may be mapped and applied in several ways. For example, through :

1. The use of oriented-object composition operators like composition or delegation, etc.
2. The use of solution based on code weaving. This consists of the fusion of the views code according to the coordination description. Another elegant way to reach this goal is to map coordination rules into Subject Oriented

<pre> // The first view class PCV{ PCV_caller caller ; PCV_caller definer ; PCV_caller glue; ... } /*-----*/ class PCV_caller{ call(){ glue = getGlue(); return (glue.call()) } } /*-----*/ class PCV_glue{ call(){ definer = getDefiner(); return (definer.call); } } /*-----*/ class PCV_definer{ call(){ Ref = refPortConnectedToThisRole(); return (Ref.method()); } } /*-----*/ </pre>	<pre> // The second view class SV{ SV_token token ; SV_computation computation ... } /*-----*/ class SV_token { int acquire(){ Ref = refRoleConnectedToThisPort(); Ref.acquire(); }; int release(){ Ref = refRoleConnectedToThisPort(); Ref.release(); }; } /*-----*/ class SV_computation { void computation { token = getToken(); token.acquire() token.release() } } /*-----*/ </pre>
---	---

Table 1: Views at implementation-level



Programming composition rules(much, override, ..) or into Aspect Oriented Programming composition mechanisms (before, after, ..).

3. The use of event-based composition framework. This solution has the advantage of being very close to the architectural description.

Our approach is based on a code weaving. According to coordination scenario, the produced code for the connector SPC can be written as follows:

```

// The SPC code
class SPC{
  // all ports and roles
  // declarations
  ...

  SV_computation_glue SPC_behavior;
  ...
}
/*-----*/
class SV_computation_glue {
  // glue after weaving with the 2nd
  // view computation
}

// ...
void glue {
  // firstly acquire the token
  token = getToken();
  token.acquire()
  definer = getDefiner();
  result = definer.call ;
  // release the token
  token.release()
  // and return the result
  return ( result);
}
/*-----*/

```

Table 2: SPC at implementation-level

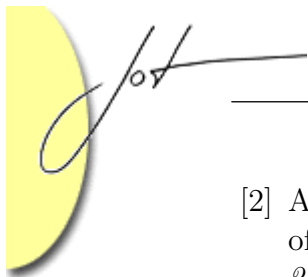
It is necessary that, for any mapping approach we must be able to map any coordination action into implementation. Nevertheless, the mapping of view description must not lose any information inherent in coordination description. For this reason, we opted for direct mapping between architectural-level and implementation one.

5 CONCLUSION

In this paper, we have dealt with the mapping between multiviews description of a given architecture and implementation. Our mapping approach is based on code weaving and fusion. At present, we generate code in manual fashion. However, to enable correct and consistent mapping, code generation tool is needed. This will be developed in our future works.

REFERENCES

- [1] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.



- [2] Abdelaziz Gacemi, Abdelhak Seriai, and Mourad Chabane Oussalah. Separation of concerns at software architecture level via multiviews description. In *The 2004 IEEE International Conference on Information Reuse and Integration (IRI-2004)*, 2004.
- [3] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCIS*, pages 498–512. Springer, 2000.
- [4] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993.
- [5] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [6] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Softw. Eng.*, 20(10):760–773, 1994.
- [7] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.
- [8] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4):314–335, 1995.