A Generic Framework for Integrating New Functionalities into Software Architectures

Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien

LIFL, INRIA project Jacquard
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq, France
{waignier,lemeur,duchien}@lifl.fr

Abstract. Integrating new functionalities into a software architecture is necessary when the application must evolve to cope with new context and user requirements. The architect has thus to manually modify the architecture description, which is often tedious and error prone.
In this paper, we propose FIESTA, a generic framework for automatically integrating new functionalities into an architecture description. Our approach is inspired by TranSAT, an integration framework. However, TranSAT is dedicated to a specific architecture description language (ADL) while our approach is ADL-independent. To do so, we have performed a domain analysis, studying for many ADLs how to integrate new functionalities. Based on our domain analysis, we have defined a generic ADL model to manipulate and reason about architectural elements that are involved in integration. Furthermore, we have defined high-level abstractions to describe different kinds of integration. Finally, we have developed a generic integration engine.

1 Introduction

Software architecture is an abstract specification of a system structure. It enables the software architect to identify the various components of the system, their interfaces and to reason about their assembly, without having to consider implementation details. Many architecture description languages (ADLs) have been proposed to represent various system properties [14].

An architecture description facilitates the comprehension, the analysis and the prototyping of a system, as well as being a good basis for its evolution. A system may need to evolve to take into account new functionalities in order to better address the application environment and user requirements. Nevertheless, adding a new functionality implies that the architect has to manually modify the architecture description. These modification operations are often low-level, tedious and prone to error, particularly in the case of cross-cutting functionalities that require invasively modifying specifications in several files.

The TranSAT framework proposes an approach to automatically integrate new functionalities into a software architecture description [5, 6]. This approach, inspired by Aspect Oriented Programming [12], relies on both the definition of

software patterns and the use of a *weaver* to automatically integrate patterns into the target architecture. A software pattern consists of three parts: a new plan, a join point mask and a set of transformation rules. A new plan is a component assembly corresponding to a given functionality. The join point mask expresses properties that the target architecture must satisfy for the integration to be possible. Finally, the set of transformation rules specifies the operations that the weaver has to perform to integrate the new plan into the target architecture. TranSAT is however dedicated to a single ADL, SafArchie [4], which limits its applicability.

In this paper we present FIESTA, a Framework for Incremental Evolution of SofTware Architectures. FIESTA entirely revisits the TranSAT approach in order to be independent of any specific ADL, keeping only the general ideas of defining software patterns and using a weaver to perform the integration. To do so, we have performed a domain analysis to understand the integration process in architectures described in various ADLs. This analysis has allowed us to identify the common architecture elements that are involved in integration, leading to the definition of a generic ADL model. This analysis has also enabled us to define more abstract expressions to specify a join point mask and the transformation rules. Overall, the FIESTA framework assists the architect during the integration process. Furthermore, our framework is built so that it can easily support new ADLs.

The rest of this paper is organized as follows. Section 2 describes our domain analysis and Section 3 presents our framework, FIESTA, including our definition of software patterns, our generic ADL model and our weaver. Section 4 describes some related work and Section 5 concludes and provides some future work.

2 Domain Analysis

In our domain analysis we have focused on the problem of adding new functionalities to the Comanche Web server. Comanche is provided as an example by Julia, the Java implementation of the Fractal component model [7]. We have manually integrated several new functionalities into this application, such as caching, logging, encryption/decryption and authentication. Furthermore, to understand the integration process for different ADLs, we have performed these integrations not only in the context of Fractal ADL [8] but also in the context of CCM [15], Olan [3], SOFA [11] and SafArchie [4]. This domain analysis has led us to identify, for each of these ADLs, the elements involved in an integration.

We have also studied other ADLs, such as Unicon [16], Darwin [13], Wright [1], AADL [2] and Acme [10]. Overall, we have studied 15 ADLs and from our domain analysis, we have established a common vocabulary to abstract architectural elements that are involved in an integration process. Furthermore, we have determined that adding a new functionality can be performed through two categories of integration.

2.1 Common Vocabulary Across ADLs

We have identified six architectural elements that the software architect needs to reason about when integrating a new functionality into a component-based architecture description: component, connector, configuration, communication point, communication element and role. We define them as follows:

- A *component* is a computational element or a data store of a system;
- A *connector* describes the interactions between components;
- A *configuration* represents an assembly of components and connectors, which corresponds to the structure of the application;
- A *communication point* is an element of a component that enables the component to communicate with its environment, *i.e.,* the other components;
- A *communication element* corresponds to what is exchanged through connectors between components;
- A *role* is a participant of the interaction represented by the connector.

Furthermore, we distinguish primitive components from composite components, as well as direct communication points from delegated communication points. A primitive component is a basic computational element and a composite defines a given assembly of primitive and composite components. A direct communication point is the source or the sink of communication. A delegated communication point propagates communication elements toward another communication point.

Table 1 illustrates how these six elements map to the specific elements of Fractal ADL, CCM, SafArchie, SOFA, Unicon and AADL. We have chosen these ADLs because they are representative of various characteristics that can be found in many ADLs.

As shown in Table 1, the concepts of primitive and composite components exist in numerous ADLs but some, such as CCM, only provide primitive components. Connectors are often simple bindings, but they can also be dedicated software entities, as in SOFA and Unicon. For example in SOFA there are three predefined connectors, CSProcCall, EventDelivery, and DataStream, and the user may also define new connector types. The concept of configuration may be explicit as in AADL and CCM, or implicit, *i.e.,* the global structure of the application corresponds to the most outwards (*i.e.,* higher-level) defined composite.

A communication point is often called interface or port, depending on the ADL. Furthermore, if the ADL is service-oriented, an interface or a port may be qualified as client or server, which is equivalent to declaring them as required or provided, respectively. In the case of a data-oriented ADL, the direction of the dataflow may be indicated, *e.g.,* in AADL, in (resp. out) specifies that the dataflow is coming in (resp. out) of the communication point. Other kinds of communication points can be found as in Unicon where communication points are players, 14 player types are defined in total.

A role corresponds to the access point of a connector. For example, in Fractal and CCM, there are two roles, one at each endpoint of the connector, one endpoint being client and the other one server. There is also one role at each endpoint of SafArchie connectors, but they have no name. The number of roles

ADL	Component	Connector	Configuration	Communication point	Role	Communication element
Fractal	primitive, composite	binding	composite	client/server interface	client/server	synchronous method call
CCM	primitive	binding	component assembly descriptor	port (Facet, receptacles, event source, event sink)	client/server	asyn/syn method call + syn event
SafArchie	primitive, composite	binding	composite	port	endpoints	synchronous method call
SOFA	primitive, composite	software entity	composite	required/provided interface	required/ provided	asyn/syn method call
Unicon	primitive, composite	software entity	composite	player	caller, definer, participant, load, *etc.*	syn method call, asyn event, syn typed flow
AADL	primitive, composite	binding	system	in/out data port, event port, event data port	input/output	synchronous typed flow

Table 1. Common architecture elements

may not be limited to two as illustrated by Unicon, which provides 11 predefined roles.

Finally, a communication point gathers a set of communication elements. Communication elements differ across ADLs. In the Java implementation of the Fractal component model, communication is performed through calls to methods that are defined in interfaces. Method calls may be synchronous or asynchronous. In SafArchie, methods are further declared as required or provided, indicating whether the component requires or provides the operation to function. Consequently, in SafArchie a communication port is not declared as required (resp. client) or provided (resp. server) as it may contain some required methods as well as provided ones. Communication elements may also be for example events or typed flows.

2.2 Two Categories of Integration

When performing our domain analysis we have identified two categories of integration. An integration may correspond to adding a new connector or require an existing connector be modified, or both. To illustrate these two categories, we consider the manual integration of a logging and a caching functionalities into the Comanche Web server application. The examples are specified in Fractal.

A high-level representation of the structure of the Comanche application is shown in Figure 1. When the `Request receiver` receives a URL request from a client, it triggers the `Scheduler` to create a new thread to handle the client. The request is first forwarded to the `Request analyzer`, then to the `Request dispatcher` and finally to the `File request handler`. If the file specified by the URL is found on the filesystem then the `Resquest dispatcher` sent file back to the client, otherwise it calls the `Error request handler`.

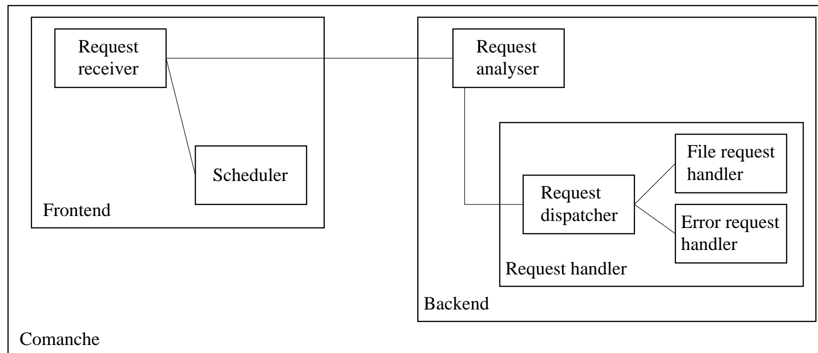


Fig. 1. Structure of the Comanche Web server

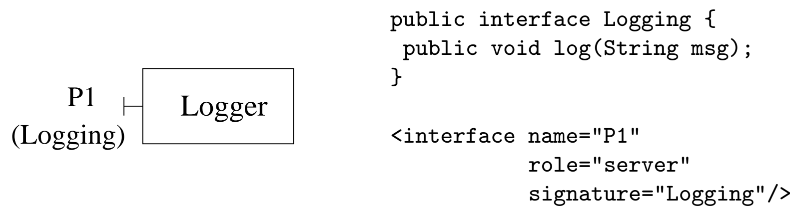


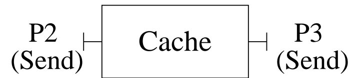
Fig. 2. Definition of the `Logger` component

Adding of a Connector: the Logging Example. We propose to add a `Logger` component that will be connected to the `Request analyzer` component in order to log all the requests sent to the Web server. The `Logger` component is shown in Figure 2. It has a communication point `P1`, which provides the `log` method through the `Logging` interface.

To integrate the `Logger` component, the architect has to add a new communication point, say `P8`, on the `Request analyzer` component. The communication point `P8` should be created so that it is *compatible* with P1. As our example is specified in Fractal ADL, compatibility implies that `P8` contains the `Logging` interface and is declared as client. To complete the integration, the `Logger` component should be placed in the `Backend` composite, and `P1` and `P8` connected using a binding.

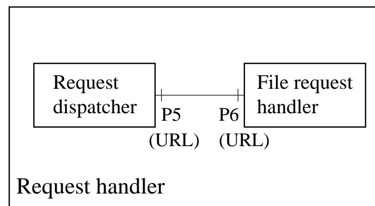
Modification of a Connector: the Caching Example. We now consider adding a `Cache` component between the `Request dispatcher` and the `File request handler` components to reduce the number of disk accesses performed by `File request handler`. This integration requires the modification of a connector.

The `Cache` component has two communication points, as shown in Figure 3. Data is received through the communication point `P2`, which offers the `Send` interface, containing the method `send`. In Fractal ADL, `P2` is a server interface.




```
public interface Send {
   public Object send(Object data);
}
<interface name="P2"
           role="server"
           signature="Send"/>
<interface name="P3"
           role="client"
           signature="Send"/>
```

Fig. 3. Definition of the `Cache` component




```
public interface URL {
   public String getPage(URL u);
}
```

Fig. 4. Excerpt definition of the `Request handler` composite

Fig. 5. Adapters

If the data passed as an argument of the `send` method is already in cache, then the cached data is returned. Otherwise, the request is forwarded through the method `send` of the communication point `P3`, which returns the requested data. This data is cached for later requests and then sent back to the caller of the `P2`'s `send` method.

To integrate the `Cache` component, the architect has to remove the connector between the `Request dispatcher` and `File request handler` components illustrated in Figure 4. The `Cache` component should be placed into the `Request handler` composite. The communication points `P5` and `P6` should be transformed so that they are *compatible* with `P2` and `P3`, respectively. To do so, adapter components may be used to perform type conversion. The adapters `A1` and `A2`, shown in Figure 5, provide this feature. In Fractal, this amounts to creating two components `A1` and `A2`, such that `A1` has a server interface `URL`, named for example `P10`, and a client interface `Send`, `P11`, and `A2` has a server interface `Send`, `P12` and a client interface `URL`, `P13`. Finally, the adapters are placed in the `Request handler` composite and connectors are created between `P5` and `P10`, `P11` and `P2`, `P3` and `P12`, and `P13` and `P6`. Figure 6 illustrates the result of the integration of the `Cache` component.

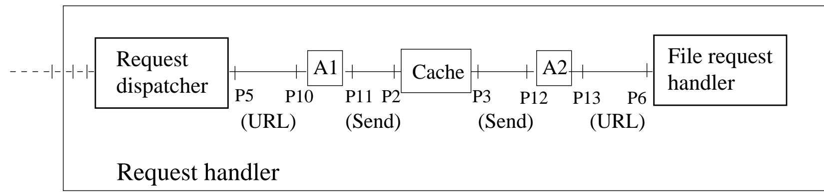


Fig. 6. Result of the integration of the `Cache` component

3 The FIESTA Framework

FIESTA is directly inspired by TranSAT. Consequently, FIESTA relies on the creation of software patterns and a weaver that both determines where a pattern can be applied in a target architecture and performs the integration. Figure 7 provides an overview of our framework.

While TranSAT is dedicated to the SafArchie ADL, our approach is decoupled from any specific ADL. Being ADL-independent has required defining new means to express software patterns, as well as the development of a generic integration engine. In this section, we present how to specify software patterns in our approach, as well as our generic architecture internal representation model and weaver, which are the key parts of our generic integration engine.

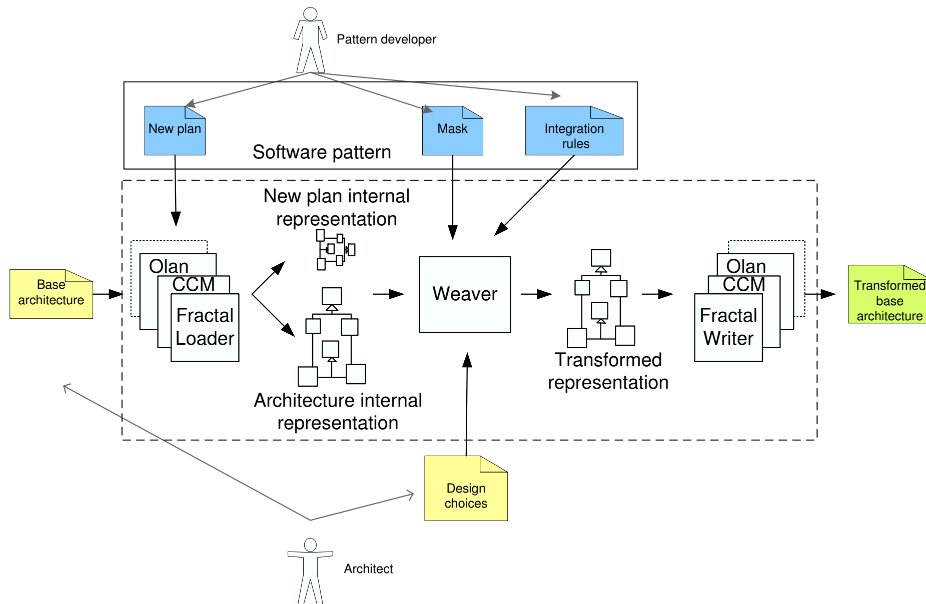


Fig. 7. Overview of the FIESTA framework

3.1 Software Patterns

A software pattern describes a given functionality and contains all the information needed to integrate this functionality into an architecture. It may be developed independently of any specific architecture by a software pattern developer, and reused on different architectures by a software architect. In our approach, a software pattern is composed of a new plan, a join point mask and a set of integration rules.

New Plan. A new plan is an assembly of components describing a functionality. For example, the `Logger` component (Figure 2) and the `Cache` component (Figure 3) can be considered as new plans. Some communication points of the assembly may not be connected yet. These points will be the points at which connectors will be attached to integrate the new plan into the base architecture, *i.e.,* the target architecture.

A new plan is described in a given ADL. It could be SafArchie, Fractal ADL or SOFA, *etc.* We have chosen to not specify the new plan with a generic ADL in order to keep all the expressiveness and the specific characteristics of each ADL, and to also not force the pattern developer to learn another ADL. The new plan will be automatically transformed into our generic architecture internal representation for use during the integration process. All the information described in the new plan will still remain in the architecture resulting from the integration.

Join Point Mask. In our approach, a join point mask expresses structural properties that the base architecture must satisfy for the integration to be possible. More precisely, it specifies a valid integration site as an abstract component assembly description.

A join point mask is composed of component masks, connector masks and communication point masks. These masks abstract the general concepts of component, connector and communication point and thus are not specific to any ADL. Furthermore, communication point masks are constrained to be *linkable* with specific communication points of the new plan. Since the new plan is expressed in a given ADL, the resolution of the linkable property will be specific to this ADL and determined at integration time.

Figure 8 illustrates the mask to associate with the `Logger` component. This mask can match any existing component of the base architecture since the logging functionality may be added anywhere. At integration time, the architect will specify which component should be concerned by the integration of the logging functionality.

Figure 9 depicts the mask for the caching functionality. This mask specifies that in order to add caching the base architecture must exhibit two components connected through a connector. The connector may go through delegated communication points if the components are not within the same composite. The communication point masks `Pn` and `Pm` are declared to be *linkable* with the communication points `P2` and `P3` (Fig. 3), respectively.

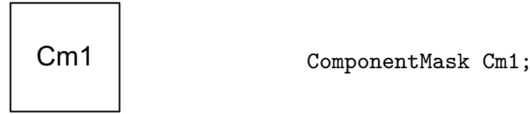
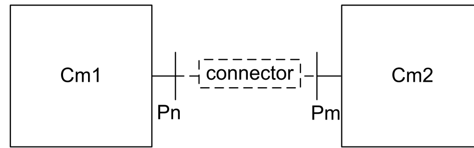


Fig. 8. Join point mask for the logging functionality




```
ComponentMask Cm1, Cm2;
CommunicationPointMask Pn on Cm1 linkable with Cache.P2;
CommunicationPointMask Pm on Cm2 linkable with Cache.P3;
ConnectorMask{Pn,Pm};
```

Fig. 9. Join point mask for the caching functionality

At integration time, the weaver will search all the integration sites satisfying the mask and the architect will specify which of these sites to affect. Each abstract name associated to a mask element will be then unified with the actual names contained in the architecture description.

Integration Rules. Integration rules are specified using integration primitives. FIESTA provides two integration primitives, one for each category of integration that we have identified during our domain analysis. We illustrate these two primitives with the logging and caching examples presented in Section 2.

Adding of a new connector in the logging example is expressed using the primitive `addConnector` as follows:

```
addConnector Logger.P1 on Cm1;
```

This rule specifies that the communication point `P1` of the new plan must be connected to the component associated with the component mask `Cm1`. This rule implies that a compatible new communication point will have to be created on `Cm1` and a connector will have to be added between this communication point and `P1`. These operations will be automatically performed by the weaver. Design choices, such as choosing the name of the new communication point will be asked to the architect by the weaver at integration time.

Modifying an existing connector in the case of the cache example is expressed using the primitive `modifyConnector` as follows:

```
modifyConnector from Cm1.Pn to Cache.P2 and
                from Cache.P3 to Cm2.Pm;
```

This rule indicates that the connector between Pn and Pm must be removed and replaced by a connector between Pn and P2 and another connector between P3 and Pm. Design choices, such as using adapters or directly modifying the communication points P2 and P3 to make the types compatible, are postponed until integration time.

3.2 The FIESTA Integration Process

Our domain analysis has shown that integrating a new functionality requires manipulating elements that are common across ADLs. Accordingly, we propose to build a generic integration framework rather than one framework for each ADL. To do so, our framework relies on a generic architecture internal representation model to express a component assembly and on a generic weaver.

The FIESTA integration process is illustrated in Figure 7. A base architecture and the new plan expressed in, say Fractal ADL, are transformed by an ADL-specific loader, each into an internal representation, according to our generic ADL model (Figure 10). The weaver transforms the architecture internal representation following the information contained by software pattern. Finally, the transformed representation is translated by a ADL-specific writer back into a Fractal ADL specification.

Generic Architecture Internal Representation Model. The model of our internal representation, which results from our domain analysis, is shown in Figure 10. To handle commonalities and variations between ADLs, our model relies on two sets of information: common structural information and ADL-specific non-structural information.

Since integrating a new functionality requires transforming the instance of the system structure at a given time, our model captures only the static structure of an architecture. The structural elements of the model are the ones identified in Section 2. Accordingly, a configuration is an assembly of components and connectors. A component may be a primitive component with direct communication points, or a composite component with delegated communication points. Communication points contain communication elements and are connected to connectors through roles. A composite component is formed of primitive and/or composite components, and connectors.

Our model takes into account the specific characteristics of each ADL and stores all the information contained in the original ADL description in order to be able to recreate it at the end. ADL-specific variabilities are stored into sets of properties, one set per structural element. These properties are represented as associations (key, value). For example, in the case of the communication point P2 of the Cache component, a property 'signature' is associated with the value 'Send' and a property 'role' with the value 'server'. Other information, such as

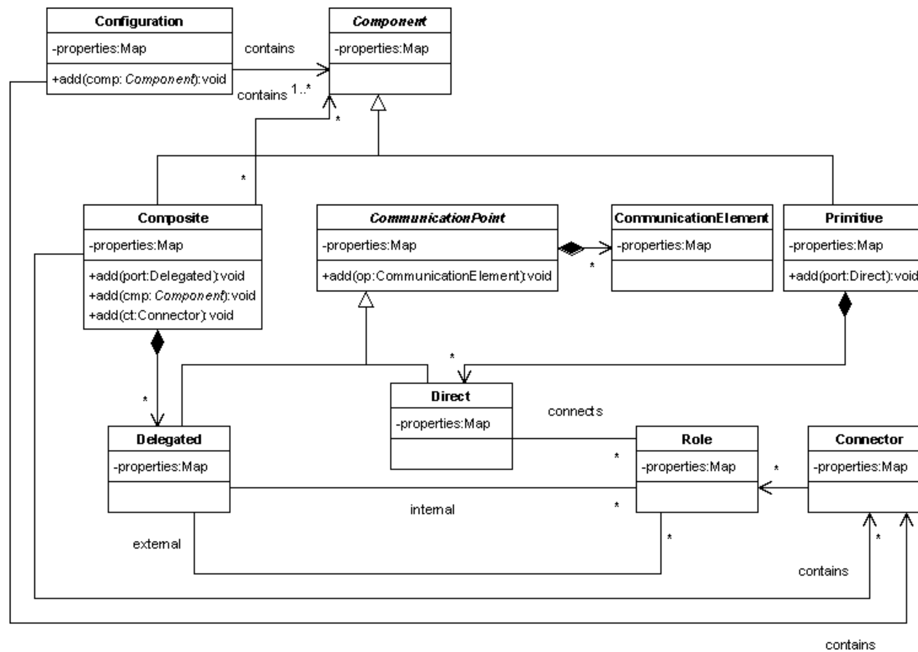


Fig. 10. Architecture internal representation model

the name of the file where an architectural element is specified in the original ADL description, is also stored.

Weaver. The weaver assists the architect during the integration of a software pattern. First, it determines from the internal representation of the architecture the set of integration sites with respect to the join point mask of the pattern. The architect chooses the sites to affect and for each site the weaver executes the integration rules defined in the pattern to integrate the new plan. Before being applied, these integration rules, which are ADL-independent, are transformed into low-level ADL-specific operations. Indeed each of the integration rules corresponds to a well-defined sequence of low-level operations, the order of the operations in the sequence being always the same, independently of the ADL used. For example, integrating the caching functionality, as described by the rule defined in the pattern, consists of removing the connector P5-P6, creating the adapters `A1` and `A2`, and finally creating four new connectors.

Though the sequence of low-level operations is generic across ADLs, these operations are performed differently depending on the ADL used. We have identified 12 low-level operations that must be specified for each ADL, such as creating a connector or determining whether two communication points are compatible or not. For example, in Fractal ADL, two communication points are compatible if they have the same signature and if one is declared as client and the other as

```
// Fractal ADL
boolean isCompatible(CommunicationPoint P1,
                     CommunicationPoint P2) {
  return
     P1.getProperty('signature').equals(P2.getProperty('signature'))
     &&
     ( ( P1.getProperty('role').equals('client') &&
         P2.getProperty('role').equals('server') )
       ||
       ( P1.getProperty('role').equals('server') &&
         P2.getProperty('role').equals('client') )
       );
}

// CCM
boolean isCompatible(CommunicationPoint P1,
                    CommunicationPoint P2) {
  return
    P1.getProperty('interface').equals(P2.getProperty('interface'))
    &&
    ( ( P1.getProperty('type').equals('facet') &&
        P2.getProperty('type').equals('receptacle') )
      ||
      ( P1.getProperty('type').equals('receptacle') &&
        P2.getProperty('type').equals('facet') )
      ||
      ( P1.getProperty('type').equals('event source') &&
        P2.getProperty('type').equals('event sink') )
      ||
      ( P1.getProperty('type').equals('event sink') &&
        P2.getProperty('type').equals('event source') )
    );
}
```

Fig. 11. Definitions of the operation `isCompatible` for Fractal ADL and CCM

server. In CCM, the notion of compatibility amounts to checking that both communication points have the same interface and that one is a facet (resp. event source) and the other a receptacle (resp. event sink). These two definitions of the operation `isCompatible` are shown in Figure 11. Another example of the operations that need to be specified for each ADL is the operation `linkableWith`, which appears in the join point mask specification and specifies under which conditions two ports may be connected with each other.

During the execution of the sequence of low-level rules, some design choices have to be made by the architect as mentioned in Section 3.1. Consequently, the weaver may ask the architect for naming new communication points, for deciding whether adapters should be created or communication points directly modified, for choosing the type of the connector, or also for deciding in which composite a new component should be placed. The number of design choices depends directly of the ADL used as some ADLs offer more possible actions than others. Nevertheless, the architect can also specify default design choices in a file so that he will not be prompted during the integration process.

4 Related Work

The TranSAT approach [5, 6] has introduced the concept of software pattern to modularize a given functionality and specify under which conditions the functionality can be integrated into an architecture description as well as the associated transformation rules to perform the integration. Our approach is also built on the concept of software patterns but is more generic since our join point mask specification and integration rules are decoupled from the SafArchie ADL or any other ADL. Furthermore, we provide higher abstractions to express how to integrate a new functionality than the TranSAT transformation rules, which remain low-level. Indeed, in TranSAT the pattern developer must express the sequence of all the operations to perform while our approach proposes two high-level integration primitives that are automatically expanded by our weaver into a sequence of low-level operations. This reduces the risk for errors on the part of the pattern developer and simplifies the verification of the coherence of the software pattern. However, TranSAT handles SafArchie behavioral descriptions while our current approach only considers structural information.

Some works, such as Acme [10] or xADL [9], have proposed a generic ADL model to represent different ADLs. Acme focuses on high-level structural properties of architectures and allows assemblies of components and connectors to be described. xADL is a generic ADL based on XML. It serves as a common internal representation of ArchStudio IDE, which consists of various architecture manipulation tools, such as Rational Rose or Armani. Consequently, xADL is more focused on representing types of architectures. Though these two models have commonalities with ours, they do not provide enough details about communication between components. For example, they do no offer the notion of communication elements and direct or delegated communication points, which are required for our purpose of integrating new functionalities.

5 Conclusion and Future Work

We have presented FIESTA, a generic framework that enables the integration of new functionalities into an architecture description. Our approach is built on a domain analysis that has led to the identification of the architectural elements involved in an integration process, independently of the ADL used, which has enabled the definition of a common ADL model. Our approach is thus decoupled from any given ADL. Furthermore, we provide higher-level abstractions to describe the join point mask and transformation rules of a software pattern than the TranSAT approach, simplifying the task of the pattern developer. Our weaver assists the architect during the integration of new functionalities. He only has to decide which integration sites to affect and to provide information related to design choices.

Our integration engine is generic in that it always performs the same actions in the same order for every ADL used. However, these actions are decomposed into sequences of well-identified low-level operations, which are implemented

differently for each ADL. These operations have thus to be specified once and for all, for each ADL that the engine supports. Adding support for a new ADL can be obtained by defining the 12 ADL-specific operations that we have identified. From our experience, most of these operations are straightforward to specify. Furthermore, a loader and a writer must also be developed, however, they can be most of the time adapted from the existing tools associated with this ADL.

Currently, we have defined the low-level operations for several of the ADLs we have studied and have fully implemented our framework for Fractal ADL. In the near future, we plan to handle more ADLs, such as SafArchie. This will require our model and rules to be extended in order to capture and manipulate SafArchie behavioral information. Furthermore, we would like to apply our approach to the development of more applications to better assess our proposition. Finally, we are interested in investigating the possibilities to adapt our approach in order to provide functionality integration capabilities at runtime.

References

1. R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, Jan. 1997. Issued as CMU Technical Report CMU-CS-97-144.
2. AS-2 Embedded Computing Systems Committee SAE. *Architecture Analysis & Design Language (AADL). SAE Standards nAS5506*, Nov. 2004.
3. R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury. Architecturing and configuring distributed application with olan. In *Proceedings of the 1st IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 241–256, The Lake District , UK, Sept. 1998. Springer-Verlag.
4. O. Barais and L. Duchien. SafArchie studio: An ArgoUML extension to build safe architectures. In *Architecture Description Languages*, pages 85–100. Springer-Verlag, 2005.
5. O. Barais, L. Duchien, and A.-F. Le Meur. A framework to specify incremental software architecture transformations. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'05)*, pages 62–69, Porto, Portugal, Sept. 2005. IEEE Computer Society.
6. O. Barais, J. Lawall, A.-F. Le Meur, and L. Duchien. Safe integration of new concerns in a software architecture. In *Proceedings of the 13th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS'06)*, pages 52–64, Potsdam, Germany, Mar. 2006. IEEE Computer Society.
7. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in Java. In *Proceedings of the 7th International Symposium Component-Based Software Engineering (CBSE'04)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22, Edinburgh, Scotland, May 2004. Springer-Verlag.
8. E. Bruneton, T. Coupaye, and J.-B. Stefani. *The Fractal Component Model*, Feb. 2004. Online documentation http://fractal.objectweb.org/specification/.
9. E. M. Dashofy, A. V. der Hoek, and R. N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103, Washington, DC, USA, 2001. IEEE Computer Society.

10. D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research Processing (CASCON'97)*, pages 169–183, Toronto, Ontario, Canada, Nov. 1997.
11. T. Kalibera and P. Tůma. Distributed component system based on architecture description: The SOFA experience. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE : Confederated International Conferences CoopIS, DOA, and ODBASE 2002*, volume 2519 of *Lecture Notes in Computer Science*, pages 981–994, London, UK, 2002. Springer-Verlag.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.
13. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989, 1995.
14. 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, Jan. 2000.
15. Object Management Group. *CORBA Component Model, v3.0, formal/2002-06-65*, June 2002.
16. G. Zelesnik. *The UniCon Language Reference Manual*. School of Computer Science Carnegie Mellon, Pittsburgh, Pennsylvania, May 1996.