Dynamic Validation of Software Architectural Connectors*

Fernando Asteasuain[1], Claudio Graiño[2], and Manuel Dubinsky[1]

[1] Universidad Nacional de Avellaneda, Dpto. Tecnología y Administración, Ing. en Informática, España 350, BsAs Argentina
[2] Dpto Computación-FCEyN,UBA, Ciudad Universitaria CABA, Argentina

Abstract. In this work we present an approach to dynamically validate the usage of software connectors in the context of software architectures. By employing aspect oriented techniques the system's execution is monitored in order to obtain an architectural view describing how processes communicate and interact with each other. This output can later be compared to the connectors specified in the architecture document to validate the consistency between the architecture specification and the implementation of the system. A case study is presented showing the potential of the approach. We believe the results are promising enough to consider future extensions including other architectural elements beyond connectors.

Keywords: Software Architectures, Dynamic Validation, Software Connectors, Aspect Orientation

1 Introduction

Over the past years the specification of software architectures has become a crucial activity for medium and large software systems. In few words, a specification of a software architecture for a given system provides a high level view of its main components and artifacts, the way they relate to each other, and the expected behavior for such interactions [16, 10]. In this sense Software Architectures can be seen as a bridge filling the gap between requirements elicitation and the resulting code [10].

One of the main challenges when dealing with software architectures is to determine whether a certain implementation of a given system satisfies its architecture specification [17, 8, 9]. There are two main reasons for this. On one side, traceability betweens architecture elements and code is most of the times fuzzy, complex and hard to achieve, since different levels of abstraction coexist simultaneously [17]. On the other side, software architectures suffer from a problem widely known as *drift and erosion* [20]. This happens when the software

* This work was partially funded by UNDAVCYT 2014, PAE-PICT-2007-02278:(PAE 37279), PIP 112-200801-00955 and UBACyT X021.

architecture specification of a system gets outdated with respect to the actual system implementation, mainly due to software changes that are not properly documented.

Some approaches tackling this problem aim to assure consistency between a system architecture's specification and its implementation by *construction* [3, 18]. However, they can only be applied only if the specific tools giving support to the approach can be employed. In some occasions, for example, it is not possible to express all the interactions of a system since they might use architectural styles which are not available on the tools to be used. Other alternatives addressing software architecture validation against a specification take a two step process [21, 15]. They first recover, either statically or dynamically, the architecture of the system. The second step consists of comparing the obtained result against the architecture specification. From these alternatives the dynamic reconstruction of software architectures has been pinpointed as the most challenging one [17, 21].

Trying to understand the architectural behavior of a system from a static perspective can sometimes be problematic since processes and other dynamic structures cannot be easily mapped to static structures. What is even more, some architectural elements exist only while the system is running (for example, a server dedicated connection to a client). In addition, there are several aspects that make dynamic architecture reconstruction appealing, specially since it includes dealing with the abstraction gap between architecture and code [21]. A first aspect to be mentioned is that the creation and behavior of a certain architectural element may involve a complex interaction between static code elements. Besides, there might be different implementations of a given architectural element. The same *Publish and Subscribe* component might be implemented using buffers, linked lists, or a customized data structure, where each option leads to different implementation code. In order to handle all the aspects some concessions have to be made. For example, the tool *DiscoTect* [17, 21] takes as input the code of the system and extracts the architecture while monitoring the system's execution. It has been widely applied since it can detect architectural components, connectors, roles, and interfaces. However, the input code must follow a certain style and naming conventions, and this restriction might be a hard to satisfy in certain contexts.

The work we present in this paper uses aspect-oriented techniques [14] to reconstruct the architecture of a system based on its execution. In particular, we focus on the utilization and validation of architectural connectors. The most relevant architectural view to reflect the dynamic behavior of a system is called *Components and Connectors view* [7, 4]. In this view, connectors play a crucial role since they establish *when*, *how* and *under which conditions* two or more components interact. Given this context, our approach answers the following question: **Is the implementation of a system communicating the way it is specified by the connectors in the Components and Connectors view?** Since it is based on code annotations, our approach does not impose any restrictions on the code. Nonetheless, it must be seen as an initial exploratory step since it only focuses on connectors, leaving out other architectural elements

as components, ports, roles or interfaces. However, we believe that the obtained results are promising enough to consider possible extensions to cope with more architectural elements.

1.1 General description of the approach and contributions

Our approach monitors a system's execution and builds an architecture view specifying how the detected connectors are used by the running processes. This is achieved using aspect orientation, a modularization technique that is suitable for runtime system monitoring. Using aspect orientation the functioning of a system can be interrupted at certain execution points in order to introduce the behavior of an aspect. The definition of an aspect includes the specification of its behavior and the identification of those places of the system that trigger its application. In our approach aspects are in charge of observing the execution and detecting the presence of architectural connectors. With the information gathered by the aspects our tool builds an architectural view showing what components exists in the system and how they interact with each other. The tool was implemented using *AspectJ*, perhaps the most popular aspect oriented programming language. As AspectJ is an extension of the Java programming language, our tool only works with applications written in Java. However, we believe the approach could also be implemented in other aspect oriented programming languages.

In order to accomplish their task aspects assume that the code implementing the system is properly annotated indicating those places in the code where the connectors are defined and used. The usage of code annotations is not new and has been largely used in the past years as a way of building a higher level of abstraction and introducing a more robust layer to interact with than code itself [19, 12]. In a software architecture domain this is particularly interesting since it helps to reduce the gap between architectural elements and code. One classic problem of code annotations is how to properly annotate the code, specially in those cases where there is little knowledge of the code implementing the system. We alleviate this issue by enabling the possibility of an incremental and localized annotation process. This is addressed in section 3.

The rest of the paper is structured as follows. Section 2 details the connectors our tool can detect, how they are recognized by the aspects and how our tool builds the architectural view. Section 3 discusses some important topics related to our approach whereas section 4 illustrates our tool in action by analyzing a case of study. Section 5 briefly discusses related work and section 6 presents conclusions and future work.

2 Selecting, Specifying and Detecting Architectural Connectors

In this section we describe the type of connectors our approach deals with and we specify the protocol and expected behavior for some of them. Finally, we explain the process to detect the connectors using aspect-oriented techniques

and describe how the architectural view is built. A simple example is shown to illustrate this process.

2.1 Selecting Connectors

Connectors play a crucial role in any software architecture specification since they dictate how the different parts of the system communicate with each other. In particular a connector allows to express how information and data progress and flow through the system and which protocols are used. In the literature there are currently available a plethora of different taxonomies describing connectors' properties and behavior [4, 7, 20]. Taking this into account, we believe it is important to mention which software connectors our approach can handle and state the expected protocol for each one. The connectors detected by our approach are the following: *Asynchronous Call*, *Synchronous Call*, *Pipe*, *Publish and Subscribe*,*Client-Server*, *Router*, *Broadcast* and *Blackboard*. Despite this selection might be considered arbitrary, the items in the set allows to express the most common software interactions between two or more processes. As an example of the expressivity of the set, it is worth mentioning that it covers all the connectors used by *Red Hat* to describe the software architecture of the products of the company [1]. What is more, it would not be difficult to add new connectors into the set if necessary.

2.2 Specifying Connector's Behavior

The behavior specification of a connector is crucial since it guides the runtime detection procedure performed by our approach as it will be later explained in the next section. Since our approach is only focused in detecting connectors in runtime the specification does not need to include notions such as ports, role and other similar concepts. Section 6 mentions the possibility to include these concepts in future work. It should be noted that we only present here an initial specification due to the exploratory phase of our tool. A more formal connector's specification needs to be addressed as future work.

Due to space reasons we now only briefly discuss in this section the initial specification for only two connectors: *Synchronous Call* and *Pipe*. The complete connector's specification and detection can be found in [11]. The *Synchronous Call* connector is perhaps the most used and known software architecture connector. Roughly speaking, a process calls some subroutine from other process and waits for an answer to continue its execution. Regarding the *Pipe* connector we define its behavior as an intermediate structure communicating two processes or components: a component producing the data and a component consuming the data. We denominate these actions as: *push* (writing in the pipe) and *pop* (reading from the pipe).

2.3 Detecting Connectors in Runtime and Building the Architectural View

In this section we describe how using aspect-oriented techniques our approach dynamically detects which connectors are being used based on the system's execution, and how it builds the architectural view. We define an aspect for each available connector. Each one of these aspects will be in charge of detecting the presence of a given connector. As it it previously mentioned, the tool assumes that the source code is annotated in those places implementing the protocol of each connector. This is true for every connector excepting the *Synchronous Call* connector. Our tool considers any method call without annotations as two components communicating with a *Synchronous Call* connector. Based on the annotations, the aspects can infer the presence of a given connector. For example, the next code fragment sketches part of the definition of the *Pipe* Aspect (see Listing 2-1). In particular, the code fragment describes those moments where the aspect should intervene: whenever a certain object invokes a method annotated as *PipePush* or *PipePop*. Note that the annotations name and quantity follow the connector's specified protocol. The complete aspect definition for each connector can be found at [11].

Listing 2-1. Part of the Pipe Aspect Definition

```
1 call(@PipePop * * (..));
2 call(@PipePush * *(..));...
```

Based on the information gathered by the aspects there exists a central process named Architectural Builder who builds the connectors view. This process keeps track of the interactions among components, and the connector used in each interaction. Since all the aspects are observing the system's execution at the same time we define an aspect's application precedence in order to guarantee that the architecture view is properly built. For example, to avoid identifying a method call to a pipe structure as a *Synchronous Call* connector instead of a *Pipe* connector. This is related to the Aspects Interference problem [5], and it is later discussed in section 3.

The architectural view is updated each time new information is obtained by any of the aspects. We now present a simple example to illustrate the detection of connectors in runtime and the process in charge of building the architecture view. The reader is referred to [11] for more details about the Architectural Builder process.

A Simple Example Suppose a system implementing two components communicating through a *Pipe* Connector. More concretely, an *EmailsPipe* class implementing a pipe, and two components using it: the *EmailCreationGUI* and the *EmailProcessor* class. In this context, the expected output for the tool would be a view showing that these classes are communicating through a *Pipe* connector.

The code fragment in Listing 2-2 shows the definition of a *EmailsPipe* class where two of its methods (*pushNewEmail* and *popNextEmail*) are annotated as

implementing a *Pipe* connector's protocol. The annotations are shown in lines 2 and 6.

Listing 2-2. A Class Implementing a Pipe

```
1 class EmailsPipe {
2 @PipePush
3 public void pushNewEmail(Email email){
4 emails.add(email);
5 }
6 @PipePop
7 public Email popNextEmail(){
8 emails.getFirst();...
9 }
```

Similarly, the next code fragment (Listing 2-3) shows part of the code for the two classes of the system communicating through the pipe: the *EmailCreationGUI* and the *EmailProcessor* class.

Listing 2-3. Implementation of the Components Using the Pipe

```
1 class EmailCreationGUI {
2 public void newEmail(Email emailReceived){
3 emailsContainer.pushNewEmail(emailReceived);
4 }
5 }
...
6 class EmailProcessor {
7 public void processEmail(){
8 EmailPipe emailToProcess=emailsToProcess.popNextEmail();
9 // ...
10 }
11 }
```

When the *pushNewEmail* is invoked (shown in line 3 in Listing 2-3) the *Pipe* aspect enters in the game since a method annotated as *PipePush* is called. The pipe aspect collects the information, which is in turn passed to the Architectural Builder which starts to build a *Pipe* relationship between the class *EmailCreationGUI* and a *Pipe* connector. The architectural builder does not have at this point enough architectural information to fully establish a pipe connector since no objects have consumed from the pipe. In other words, no pop annotated method has been invoked yet. However, it is registered that the class *EmailCreationGUI* performed a push over a pipe. It is worth noticing at this point that the aspect in charge of detecting synchronous call connector will also be activated. However, since this method invocation has been previously analyzed by the *Pipe* aspect the *Synchronous Call* aspect ignores this method call. Recall that there exits a precedence rule that dictates which aspect is applied first.

Eventually, the *popNextEMail* method is invoked (see line 8 in Listing 2-3). When this invocation occurs, the pipe aspect gathers this information and

the Architectural Builder updates the view establishing that classes *EmailCreationGUI* and *EmailProcessor* communicates through a pipe connector as it was expected. It can establish this relationship since an object of class *EmailCreationGUI* performs a push action over a pipe, and a object of class *EmailProcessor* performs a pop action over the same pipe.

3 About Aspects Precedence, Incremental and Localized Analysis and some Extra Features

In this section we highlight some important points of our approach. We first analyze some decisions regarding aspects precedence, which are related to a crucial problem for the aspect-oriented community such as the Aspects Interference Problem [5]. In second term, we describe how by employing an special type of annotation our approach is suitable for an incremental and localized architectural analysis. Finally, we present some extra features available in our tool beyond the discovery of software connectors.

3.1 Aspects Precedence and the Aspects Interference Problem

The Aspect Interference problem [5] is a very well known problem in the aspect oriented community. This problem occurs when two or more aspects can act on the very same point of interest, such as a method call. In these cases, it is important to resolve questions like: Which aspect should be applied first? Why? Does it matter? In particular, this problem is exacerbated if the correct behavior of the system depends on the order in which the aspects are applied.

In our case this problem occurs when two or more of the aspects defined to identify connectors interact within the same method call. For example, a method call could be registered either as a synchronous call or as a part of a pipe behavior. In order to tackle this problem we define a particular precedence of aspects application, so that the tool analyzes each particular method call in the right order. After a rigorous analysis we define the following precedence: *Pipe* , *Publish Subscribe*, *BlackBoard*, *Client Server*, *Broadcast*, *Router*, *Asynchronous Call* and finally, *Synchronous Call*. This implies that the *Pipe* aspect will be always executed first (it has the highest precedence) and the *Synchronous Call* aspect will always occur in the last place (a simple method call will be catalogued as a *Synchronous Call* connector if no other connector was previously detected). Getting back to the previous example when trying to distinguish between a *Pipe* connector or a *Synchronous* call, if the method call was part of a pipe structure the resulting architectural relationship will be registered as a *Pipe* as expected since the *Pipe* aspect has higher precedence than the *Synchronous Call* aspect.

One interesting final remark regarding aspects interference is about the expressivity of the language used to specify aspects' behavior. We would have needed to specify aspect application as follows: "Only apply this aspect at this execution point if and only if no other aspect has been applied here before". Similarly, work in [6] proposes a richer aspect model where the user can specify

this exclusive application of an aspect at a certain point. Since the language we used to implement our approach (AspectJ) does not support this kind of expressions, this was solved in an ad-hoc fashion, keeping a structure of the points of interest already visited. Under this perspective, we advocate for aspect oriented languages implementing a richer model to express and specify aspect's behavior.

3.2 Incremental and Localized Architectural Analysis

By defining a special type of annotation our approach is able to allow an incremental and localized architectural analysis. In this sense, we introduce an special annotation named "Ignored" pursuing two main purposes. On one side, some methods might be known as not being relevant for architectural analyses. In those cases, they can be annotated as "Ignored" so that aspects can simply ignore their invocation. The second objective for this annotation is to allow an incremental and localized construction of the architectural view. For example, if only a certain portion of the code is to be addressed or only a particular interaction between two or more components need to be validated the rest of the implementation can be marked as ignored so that the tool can only focus on the exact portion of the system that is relevant at that given moment. This is also particularly interesting since it allows the possibility of an incremental discovery of the architecture. The user might start analyzing only a small portion of the system and later expand the area covered by our tool in an incremental flavour by simply removing the ignored annotation. This incremental process is also helpful to properly annotate the code of the system if there is little knowledge of the system behavior. The user of the tool can initially annotate only a portion of the code restricting the analysis to that portion, instead of trying to annotate the whole code at once.

3.3 Some Extra Features

The current state of our tool is able to provide two more interesting features besides the dynamic discovery of software connectors. In the first place, it can detect a more deeper analysis related to the *Publish Subscribe* connector. In particular, in can detect what type of messages is receiving each subscriber. This information is helpful in order to validate that each component is receiving the data is supposed to receive and nothing else. This is achieved by recording not only the components interacting at a given point but also the type of the messages exchanged in the interaction.

More related to an architectural analysis, our tool can suggest the presence of a *Pipe and Filter* architectural style and not only the presence of a pipe connector. This style describes a certain interaction between two or more components communicating with pipe connectors in a sequential fashion. When collecting the information gathered by the *Pipe* aspects, the tool can build a chain of processes interacting all together with two or more pipes over the same structure, and therefore detecting not just a pipe connector but a *Pipe and Filter* architectural style.

4 Case Study

In this section our approach is shown in action by validating the architecture of a given system. Although the system under analysis is simple it features non trivial architectural behavior exhibiting the use of several type of different connectors resulting in a interesting case of study. The system, called "My Little Tomato Plant", was implemented as a final assignment of a Software Engineering course at University of Buenos Aires, Argentina. It consists of a system in charge of controlling the growth of a tomato plant. Given the information obtained by sensors attached to the plant (indicating water, light and humidity levels) the system executes the necessary actions to take care of the tomato plant and to assure that it grows healthy. These actions are obtained based on botanical knowledge and a growth plan indicating the expected health parameters of a tomato plant through its life cycle. These actions are built as orders to *actuator* components that can augment or diminish the levels of light, water and humidity that the tomato plant is receiving. Figure 1 shows the architecture specification for the system. It can be seen that several connector types are used: *Synchronous* and *Asynchronous Call*, *Publish and Subscribe*, *Pipe* and *Client Server*.

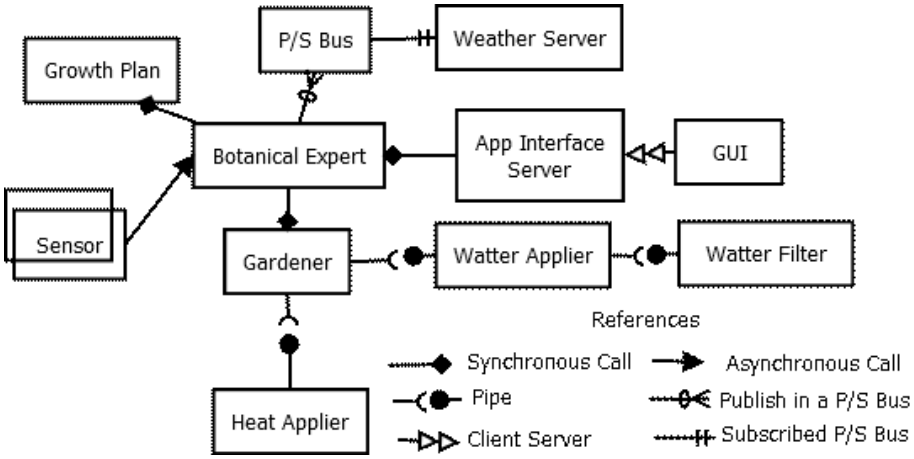

Fig. 1. Original Architecture Specification of the System

Given a certain code implementing the system our tool was employed to obtain an architectural view based on the system's execution. Figure 2 shows the architecture built by the tool.

Two main differences are appreciated when comparing both views (the architecture built by the tool in Figure 2 and the architecture original specification in Figure 1). One one side, there is a missing collaboration between two components. In the original specification there is a *Synchronous Call* relationship between the *Growth Plan* and the *Botanical Expert* component which is not

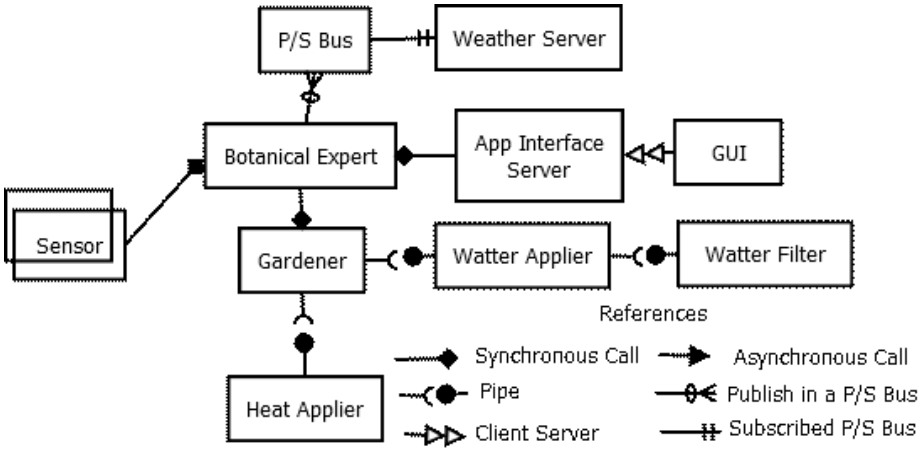

Fig. 2. The architectural view of the system built by the tool

present in the architecture built by the tool. It can be the case that either the *Growth Plan* component was marked as ignored, or that the component was not involved in the system's execution when the view was built. In these cases, the user can remove the ignored annotation, or run again the system in such a way that the *Growth Plan* is executed. If after realizing these changes the *Growth Plan* component is still missing in the view then this inconsistency between both views indicates a potential serious problem: either there is an implementation bug and the *Botanical Expert* component is never interacting with the *Growth Plan* component, or an architectural decision was made during the implementation phase and the original specification was never updated. On the other side, there is a connector mismatch between components *Sensor* and *Botanical Expert*. In the original architecture it is specified that they should interact through an *Asynchronous Call* connector whereas in the view built by the tool they interact through a *Synchronous Call* connector. A similar analysis to the one seen in the previous case can be performed: either the original specification is outdated or the current implementation is not behaving as it is supposed according to the specification.

In both cases the tool resulted indeed helpful to identify architectural behavior alarms either in the shape of errors in the implementation or specific items to update the original architecture specification. Finally, it is worth mentioning that the tool also properly identified a *Pipe and Filter* style. In addition, the output obtained by the tool was also used to validate that the subscribers processes were receiving the expected information from the publishers.

5 Related Work

Approaches in related work can be divided into three categories [21, 17]. The first one groups those alternatives which aim to assure consistency between a

system architecture's specification and its implementation by *construction* [3, 18]. These approaches work efficiently when the tools that give support to them can be actually employed [21]. In some occasions, for example, it is not possible to express all the interactions of a system since they might use architectural styles which are not available in the tools to be used.

The second category consists of those approaches based on static code analysis [15, 13]. These approaches aim to build the architecture of a system upon its code. However, they suffer from some known problems [21, 17]. Trying to understand the architectural behavior of a system from the code can be sometimes problematic since process and other dynamic structures cannot be easily mapped to the static structures reflected by the code. What is more, some architectural elements exist only while the system is running and therefore cannot be captured using these techniques. Work in [2] uses static analysis and annotations to build a runtime architectural structure where conformance analysis can be applied. The main purpose of our is different since we are only interested in building a dynamic view of the architecture. Similarly, annotations in [2] are focused in the structure and hierarchy of the system while in our work they are used to identify the behavior of the connectors. Finally, approaches in the third category focus on the extraction of a system architecture upon the dynamic observation of the system execution. Probably the most representative example of the category is the tool *DiscoTect* [21, 17]. It has been widely applied since it can detect architectural components, connectors, roles, and interfaces. However, the restriction for the code to follow certain naming conventions and other similar limitations might be hard to satisfy in certain contexts.

6 Conclusions and Future Work

In this work we present a tool that builds an architectural view based on the system's execution. In particular, it is focused on detecting the connectors used while the system is executing. The tool requires that the source code is properly annotated in those places implementing the connector's protocol. We explained how can this be done in an incremental and localized manner even in the case where there is little knowledge of the source code. We applied our tool to a non trivial example and the results showed that the tool helped to identify architectural behavior mismatches between the running system and the original specification of the system. We believe our tool constitutes a solid first exploratory step towards a runtime discovering architectural tool.

Our tool was implemented using the AspectJ language, following aspect-oriented techniques. In this sense, we found some obstacles when trying to specify the aspects behavior and we realized that a more richer language model is needed to properly address the Aspect Interference problem [5]. Regarding future work, we would like to augment our expressivity to denote architectural behavior beyond connector's detection. For example, we would like to add notions like ports, roles, styles among others, in order to become a more precise architectural tool. This next step would allow the possibility to interact with

other software architecture tools like *Arch Java* [3] or *DiscoTect* [21]. We would also like to explore the possibility to annotate the code automatically.

References

1. Enterprise integration patterns. https://www.redhat.com/es/files/resources/en-rhjb-fuse-eip-flashcards-10611447.pdf.
2. M. Abi-Antoun and J. Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In *ACM SIGPLAN Notices*, volume 44, pages 321–340. ACM, 2009.
3. J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *ICSE 2002*, pages 187–197. IEEE, 2002.
4. L. Bass. *Software architecture in practice*. Pearson Education India, 2007.
5. L. M. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Analysis of Aspect-Oriented Software (ECOOP 2003)*, 2003.
6. S. Casas, J. Pérez-Schofield, and C. Marcos. Conflicts in aspectj: Restrictions and solutions. *Latin America Transactions IEEE*, 8(3):280–286, 2010.
7. P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002.
8. J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *ASE*, pages 486–496. IEEE, 2013.
9. J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. In *ICSE 2013*, pages 901–910. IEEE Press, 2013.
10. D. Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In *SFM*, pages 1–24. Springer, 2003.
11. C. Graiño. Validación de arquitecturas a través de la programación orientada a aspectos. *Tesis de Licenciatura.*, http://www.dc.uba.ar/inv/tesis/licenciatura/2015/graino.pdf. 2015.
12. M. M. Joy, M. Becker, W. Mueller, and E. Mathews. Automated source code annotation for timing analysis of embedded software. In *ADCOM,12-18*, 2012.
13. R. Kazman and S. J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *ASE*, 6(2):107–138, 1999.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP 2001*, pages 327–354. Springer, 2001.
15. L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE software*, 27(5):82, 2010.
16. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
17. B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *TSE*, 32(7):454–466, 2006.
18. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering, IEEE Transactions on*, 21(4):314–335, 1995.
19. R. Suzuki. Interactive and collaborative source code annotation. In *ICSE*, pages 799–800. IEEE Press, 2015.
20. R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.
21. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems. In *ICSE*, aosfpp 470-479, 2004.