Dynamic Building of Software Architectural Connectors View*

Fernando Asteasuain[1,3], Claudio Graiño[2], and Manuel Dubinsky[1,2]

[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
[3] UAI-CAETI CABA, Argentina

Abstract. In this article 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 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. Two case studies are 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 [20, 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 [21, 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 [21]. On the other side, software architectures suffer from a

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

problem widely known as *drift and erosion* [25]. This happens when the software architecture specification of a system gets outdated with respect to the actual system implementation, mainly due to software changes that are not properly documented.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 specification and its implementation by *construction* [2, 22]. However, they can only be applied only if specific tools are employed. Sometimes it is not possible to express all the interactions of a system since they might use architectural styles not supported by the tools. Other alternatives addressing software architecture validation against a specification take a two step process [26, 19]. First they collect information (either statically or dynamically) of 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 [21, 26].

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. Even more so, some architectural elements exist only while the system is running (for example, a server dedicated connection to a client).

The work we present in this article uses aspect-oriented techniques [17] 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

Our approach monitors a system execution and builds an architecture view specifying how the detected connectors are used by the running processes. This is achieved using aspect orientation. 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 exist 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 [24, 15]. 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.

1.2 Previous work and new contributions

In [3] we introduced our tool describing its main features. We now upgrade that work presenting the following new characteristics:

- An extended connectors description and specification and a new connectors categorization based on their behavior. This categorization is latter used to build the connectors' view.
- A complete description of the aspect orientation technique used to dynamically capture the presence of the selected architectural connectors. This description covers all the connectors where only two of them were described in [3].
- An upgraded Building Architectural Process description. Architectural Analyzers are introduced in this work describing more thoroughly how the architectural connector view is built.
- An example showing how to proceed to obtain an incremental and localized annotation process.
- A new case of study showing that our tools do not only contemplate the mentioned connectors, but also any possible combination of them.

The rest of the paper is structured as follows. Section 2 details the connectors that our tool can detect, their protocols's specification and a proposed categorization based on their behavior. Section 3 describes how using aspect-oriented techniques our approach dynamically detects which connectors are being used based on the system's execution. Section 4 explains how the architectural connectors view is built. Section 5 discusses some important topics related to our approach whereas section 6 illustrates our tool in action by analyzing a case of study. Section 7 shows how the incremental and localized annotation process can be realized. Section 8 exhibits a case of study presenting a customized connector. Section 9 condenses some ideas behind lessons learned and threats to validity. Section 10 briefly discusses related work and Section 11 presents conclusions and future work.

2 Connectors: Which ones are detected and why

In this section we describe the connectors that are covered in our approach and the reasons behind this selection. Finally, we propose a classification of connectors based on their architectural behavior which play an important role when discovering connectors usage based on the system execution.

2.1 Set of Connectors Selected

Connectors play a crucial role in any software architecture specification since they dictate how the different parts of system communicate with each other. In particular a connector allows to express how information and data flow through the system and which protocols are used. A connector might determine, for example, if two processes interact through a simple *asynchronous call* or using instead a complex buffer based on events such as *Publish and Subscribe* [4].

In the literature there exits a vast variety of approaches enumerating and classifying software architecture connectors. In other words, there are available a plethora of different taxonomies describing connectors properties and behavior [4, 7, 25]. 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 of them. Despite this selection might be considered arbitrary, the items in the set allow to express the most common software interactions between two or more processes. As an example of the expressivity of the selected connectors, it can be mentioned that they cover all the connectors used by *Red Hat* to describe the software architecture of the products of the company [14].

The selected connectors are the following: *Asynchronous Call*, *Synchronous Call*, *Pipe*, *Publish and Subscribe*,*Client-Server*, *Router*, *Broadcast* and *Blackboard*.

This set of connectors might be classified into three different categories according to their behavior: *Direct* connectors, *Naive Intermediary* connectors and *Sophisticated Intermediary* connectors. These categories are explained in the next section.

2.2 Characterization of Connectors Behavior

The selected connectors might be grouped into three categories based on how they architecturally behave. Basically, we will distinguish between those connectors using a somehow intermediary structure or not, and wether this intermediate artifact performs an architectural relevant action or not for the connector to be detected. The proposed categories are: *Direct* connectors, *Naive Intermediary* connectors and *Sophisticated Intermediary* connectors. As it was said before, these categories play a crucial role for the detection used in our approach to reveal the usage of the connectors while monitoring the system's execution.

Direct connectors The connectors in this category relate in a straightforward manner two given processes without any intermediaries in between. We include in this category the following connectors: *Asynchronous Call*, *Synchronous Call*, and *Client-Server*. A process might directly call another (either waiting or not for a response) or invoke a given service in a server. In any case, the two processes related by the connector are the only ones involved in the communication.

Naive Intermediary Connectors The connectors in this category uses an intermediary to communicate two or more processes. They are called *Naive Intermediary* since the intermediary does not realize any other relevant architectural action (for the connector to be properly detected) besides appropriately transporting information from the source to the destination. Connectors *Pipe*, *Publish and Subscribe* and *Blackboard* belong in this category. For example, a *Pipe* connector takes the output from a process and places as the input for the following process. Similarly, a *Publish and Subscribe* connector will communicate a given event to any interested process. In both cases the intermediate structure do some tasks. However, they are not architecturally relevant for the proper detection of the connectors.

Sophisticated Intermediary Connectors Finally, connectors in this category uses an intermediary, which will take decisive actions while communicating processes. Connectors *Router* and *Broadcast* belong in this category. In this case, the intermediary structure performs relevant architectural actions which must be considered in order to properly detect the usage of the connector. For example, in the router connector, the connector forwards the information to only a subset of the possible destinations. Similarly, in a broadcast flavour of communication, the connector must iterate through all the possible destinations. In these cases these actions are architecturally relevant for the proper detection of the connector since they expose the functioning of the connector.

Table 1 summarizes the proposed categories for classifying the selected architectural connectors. In the table, *NI* stands for *Naive Intermediary* and *SI* stands for *Sophisticated Intermediary*.

2.3 Specifying Connectors Behavior

The behavior specification of a connector is crucial since it guides the runtime detection procedure performed by our approach. 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 11 mentions the possibility to include these concepts in future work.

Connector's behavior is specified using a labeled finite automaton-based notation following the one used in [25]. Transitions labeled with an exclamation mark (!) can be seen as events produced locally in the automaton whereas transitions labeled with a question mark (?) can be seen as events produced in other automaton. In this way, actions are used to synchronize automata and behavior can

Table 1. Categorization of connectors' behavior

Connector	Direct	NI	SI
Asynchronous Call	X		
Synchronous Call	X		
Pipe		X	
Publish and Subscribe		X	
Client-Server	X		
Router			X
Broadcast			X
Blackboard		X	

be denoted as the composition of two or more automatons. As an example, figure 1 presents an automaton describing the behavior of the Synchronous Call connector, which combines the behavior of the caller and the callee. Once the caller makes the call, represented by the event *CALL*, the connector forwards it to the callee (*FORWARD_CALL* event). Similarly, when the callee produces its output(*RETURN* event) the connector forwards it to the caller (FORWARD_RETURN event).

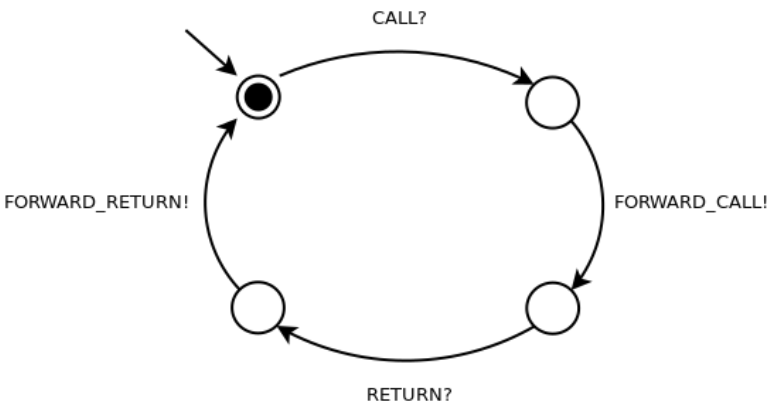

Fig. 1. An automaton describing the behavior of the Synchronous Call connector

Since in this first stage of our tool concepts like ports or roles are not needed we can abstract the behavior of the connectors in a very direct and simple way. Given this fact we can rely on a more basic specification of the connectors behavior. For example, considering the behavior of the Synchronous Call connector in figure 1 we can simply assume that any message exchanged between two objects is a *Synchronous call* connector. This is also the case for connectors involving intermediate structures such as the 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). The other connectors covered by our tool are defined in a very similar way. The complete connector's specification can be found in [11]. It is worth to point out that extending our tool to include other meaningful architectural elements as ports or roles will imply the usage of a more complex and formal specification of the connectors. This issue in particular is contemplated as future work.

3 Detecting Connectors in Runtime

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.

Using aspect orientation the execution of a system can be interrupted at certain moments in order to introduce the behavior of an aspect, allowing a more powerful and flexible modularization technique. An aspect specification includes its behavior and the identification of those places of the system that trigger the application of the aspect's behavior. 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 is 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.

The followings sections detail how aspects are defined in order to detect the presence of the architectural connectors covered by our work.

3.1 Synchronous Call

This is the most simple connector. We will assume any message exchanged between two objects is a *Synchronous Call* connector. Given this, the *Synchronous call* aspects just detect any given call between two objects. Note that no code annotation is needed for the detection of this connector. The next code fragment sketches part of the definition of the *Synchronous Call* Aspect (see Listing 3-1). This implies that this aspect will intervene in every method call between two objects.

Listing 3-1. Part of the Synchronous Call Aspect Definition

```
1 call(* *(..));
```

3.2 Asynchronous Call

Asynchronous messages are implemented in our tool by using threads. In this context, any method call made from a thread will be monitored by the Asynchronous Call Aspect (see Listing 3-2).

Listing 3-2. Part of the Asynchronous Call Aspect Definition

```
1 call(*(..))&& withincode(void java.lang.Runnable.run());
```

3.3 Pipe

A pipe connects two processes in a chain by forwarding the output from one source to the input of the next one. These two particularly actions define the *pipe* connector protocol and they must be properly captured using two code annotation: *PipePop* and *PipePush*. The pipe aspect essentially captures any method annotated with *PipePop* and *PipePush*. This is shown in Listing 3-3. This 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.

Listing 3-3. Part of the Pipe Aspect Definition

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

3.4 Router

In this case only one code annotation is needed: *route*, to indicate which method implements the behavior of the connector. There is, however, another step to make to completely define the behavior of the connector. It is necessary to identify any method call made within the router method since the receptors of these calls stands for the receptors of the router connector. The code fragment in Listing 3-4 exemplifies this behavior.

Listing 3-4. Part of the Router Aspect Definition

```
1 call(@Route * * (..))
2 call(* * (..)) && withincode(@Route public * * (..))
```

3.5 Broadcast

This aspect is analogous to the *Router Aspect*, except that the aspect annotation is named *Broadcast* instead of *Router*.

3.6 Client-Server

Two points need to be considered for this connector. On one side, the Client-Server Aspect must detect the message sent to the server by the client to establish the connection between them (line 1 in Listing 3-5). On the other side, the aspect must also be aware of the messages sent to the object representing the connection itself (line 2 in Listing 3-5)

Listing 3-5. Part of the Client Server-Aspect Definition

```
1 call(@RequestConnection * * (..))
2 call(* @ClientServerConnection *.*(..))
```

3.7 Publish-Subscribe

Annotations named *Publish* and *Subscribe* are introduced to detect the interaction to the intermediate structure of the behavior (see Lines 1 and 2 from Listing 3-6). Additionally, the *Publish-Subscribe Aspect* must detect all the messages exchanged when subscribers are notified of a new publication. In order to detect only these methods calls (and not any other call) made within the code of the Publisher a new annotation is introduced: *Notify* (see Line 3 from Listing 3-6)

Listing 3-6. Part of the Publish Subscribe Aspect Definition

```
1 call(@Publish * * (..))
2 call(@Subscribe * *(..))
3 withincode(@Publish * * (..)) && call(@Notify * * (..))
```

3.8 Blackboard

This aspect is very similar to the *Publish-Subscribe Aspect*. The Listing 3-7 shows part of this aspect's definition. Lines 1 and 2 identify calls made to add data to the repository and subscription to the repository. Line 3 targets the notifications made by the connector to its subscribers. Finally, line 4 uses a new annotation, named *Read*, to identify read actions from the repository.

Listing 3-7. Part of the BlackBoard Aspect Definition

```
1 call(@Store * * (..))
2 call(@SubscribeBlackboard * *(..))
3 call(@Notification **(..))&& withincode(@Store * * (..))
4 call(@Read public **(..))
```

4 Building the Connectors Architectural View

In this section we describe how the architectural view is dynamically built by the tool. Section 4.1 details the building process whereas section 4.2 illustrates the process with a simple example.

4.1 Architectural Builder

Based on the information gathered by the aspects there exists a central process named Architectural Builder that 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. The architectural view is updated each time new information is obtained by any of the aspects.

The Architectural Builder process is nourished by three types of software artifacts named *Architectural Analyzers*. Architectural analyzers are in charge of obtaining the information from all the aspects monitoring the systems execution and detect the presence of an architectural connector. With this information the Architectural Builder constructs the aimed connectors' view. Figure 2 illustrates the main steps of our approach.

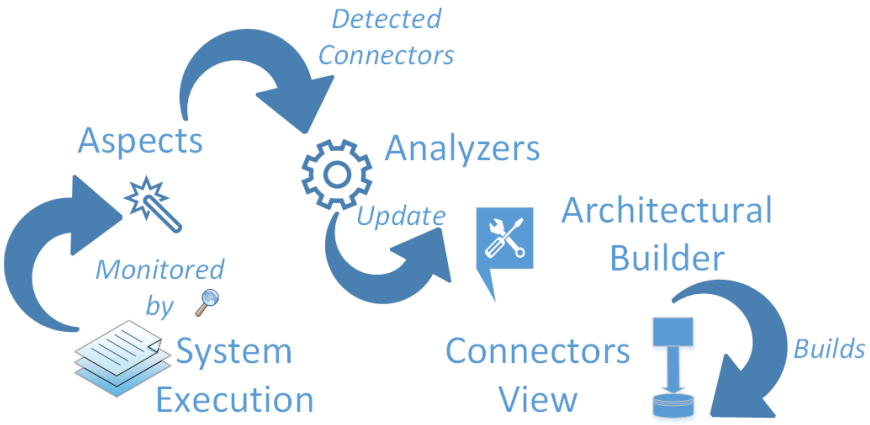

Fig. 2. Original Architecture Specification of the System

In order to perform the task they were designated for Architectural Analyzers use dynamic information about the aspects who captures a point of interest (for example, one object invoking a method from another object). This information includes the name of the aspect collecting the information, signature of the invoked method, class name of the sender and the receiver, among other items. Following the classification of the connectors' behavior in section 2 we introduce three types of *Architectural Analyzers*: *Direct*, *Naive Intermediary* and *Sophisticated Intermediary*, which are described in what follows.

A *Direct Architectural Analyzer* simple captures messages like "An object a of type A sends a message M to an object c of type C". This implies that a

Direct Architectural Analyzer manages the detection of the following connectors: *Synchronous Call*, *Asynchronous Call* and *Client-Server*.

Naive and *Sophisticated Intermediary Architectural Analyzers* are similar, but they also contemplate the flow of the execution involving the intermediate structure, such as a buffer. An *Naive Intermediary Architectural Analyzer* captures messages like "An object a of type A sends a message M to the intermediary i of type I". For example, an object placing information in a pipe invoking the pop method, or an object retrieving data from a pipe, invoking the push method. Since both messages are sent to the intermediary object we denominate these analyzers as *Naive Intermediary*. A *Naive Intermediary Architectural Analyzer* handles the information of the following connectors: *Pipe*, *Publish-Subscribe* and *Blackboard*. A *Sophisticated Architectural Analyzer* is a bit more complex, since it need to distinguish if the intermediary is the sender or the receiver of the message. For example, a router connector will only be detected if two things happen. First, a router annotated method must be invoked (in this case the intermediary is playing the role of the receiver). Secondly, a message to the expected receptors is sent by the intermediary (in this case the intermediary plays the role of the sender). So, in order to detect the architectural relation the intermediary needs to play both roles, sender and receiver, and that is why they are called actives. A *Sophisticated Architectural Analyzer* detects the presence of the following connectors: *Router* and *Broadcast*.

Since our approach monitors the system execution the architectural view construction is a continuous process and is updated as new architectural information is gathered. The main steps followed by our approach can be schematized as shown next.

- Step 1: The systems is running.
- Step 2: Aspects monitor the system's execution in order to detect the presence of connectors.
- Step 3: Certain execution point *ep* is captured by one of the defined aspects.
- Step 4: If *ep* has already been analyzed by one aspect, the execution point is discarded.
- Step 5: If *ep* has not been previously analyzed, the aspect collects the context of execution of *ep* and passes that information to the corresponding Architectural Analyzer. The *ep* is marked as analyzed.
- Step 6: The Architectural Builder analyzes the received information and updates accordingly the architectural view.
- Step 7: Once the architectural view is updated, the system continues its execution at step 1.

Since we define one aspect for each selected connector, while the system is running all the aspects monitor the execution at the same time. Some points of interest such a method invocation might call the attention of two or more aspects at the same time. An object o of type O calling a method m of an object p of type P might be a simple synchronous call between two processes or perhaps it belongs to a more complex connector like a *Pipe* and it represents one processes

performing a push over a pipe. Taking this into account, aspects analyze execution points in a certain order so as to guarantee that the architectural view is properly built. What is more, only one aspect must analyze each execution point of interest. This is checked in step 4 whereas line 5 shows when a certain execution point is marked as analyzed. The reasons for this are inherited by current limitations of the aspect oriented language used in our work, *AspectJ*.

To further clarify how our approach works, a simple example is given next. The reader is referred to [11] for more details about the Architectural Builder process.

4.2 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 4-8 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 4-8. 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 4-9) shows part of the code for the two classes of the system communicating through the pipe: the *EmailCreationGUI* and the *EmailProcessor* class.

Listing 4-9. 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 4-9) 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 a *Naive Intermediary Architectural Analyzer* which starts to build a pipe relationship between the class *EmailCreationGUI* and a Pipe connector. The mentioned analyzer 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. If it was the case that no other aspect had previously intervened, then a synchronous call connector will be established between the sender and the invoked objects. Recall that there exist a precedence rule that dictates which aspect is applied first.

Eventually, the *popNextEMail* method is invoked (see line 8 in Listing 4-9). When this invocation occurs, the pipe aspect gathers this information and the *Naive Intermediary Architectural Analyzer* can therefore detect the presence of a pipe connector. More particulary, it can establish that classes *EmailCreationGUI* and *EmailProcessor* communicate through a pipe connector, since an object of class *EmailCreationGUI* performs a push action over a pipe, and objects of class *EmailCreationGUI* performs a pop action over the same pipe. Analogously to the previous discussion, a possible synchronous call connector between *EmailProcessor* and *EMailsPipe* is discarded. This information is sent to the Architectural Builder who accordingly updates the view exhibiting two components communicating through a pipe connector as it was expected.

5 Related topics

In this section we highlight some important points of our approach. We first analyze in section 5.1 some decisions regarding aspects precedence, which are related to a crucial problem for the aspect-oriented community: the Aspects Interference Problem [5]. In section 5.2 we describe how by employing a special type of annotation our approach is suitable for an incremental and localized architectural analysis. In this sense, we define a special type of annotation used in our tool to detect the presence of architectural connectors in runtime: the "Ignored" annotation. Finally, in Section 5.3 we present some extra features available in our tool beyond the discovery of software connectors.

5.1 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.

5.2 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 a 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 *Architectural Analyzers* can simply ignore their invocation. The other 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 focuses 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.

5.3 Beyond connectors detection

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, it 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.

6 Case Study: The tool in action

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. Given these characteristics, the analyzed system results in a solid case of study to apply and validate our approach. The system, called "My Little Tomato Plant", was implemented as a final assignment of a Software Engineering course at the Universidad de 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. The system consists of nearly *5000* lines of code distributed among *35* Java classes. Figure 3 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*. Figure 3 also shows in the right corner a reference to identify each connector in the view.

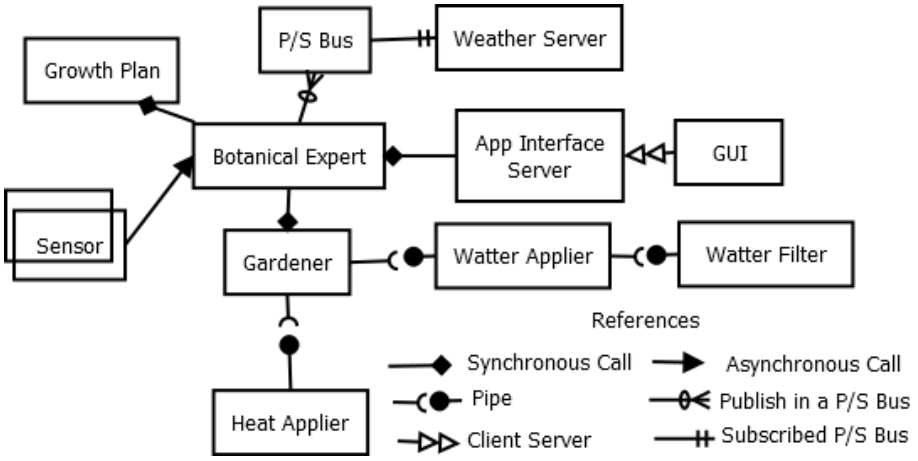

Fig. 3. Original Architecture Specification of the System

Given a certain code implementing the system our tool was employed to obtain an architectural view based on the system's execution. Figure 4 shows the architecture built by the tool relying on a fully annotated source code implementing the system.

Two main differences are appreciated when comparing both views (the architecture built by the tool in Figure 4 and the architecture original specification in Figure 3). 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 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 Ex-*

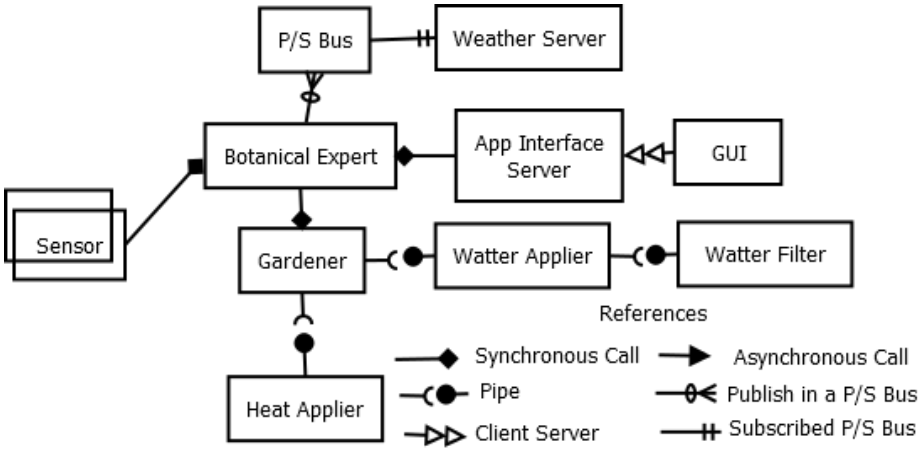

Fig. 4. The architectural view of the system built by the tool

pert. 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.

7 Incremental and localized annotation process: An example

The tool implementing our approach assumes that the source code is annotated indicating those places in the code implementing the connectors protocol. However, as it was previously mentioned in section 5, if there is little knowledge of the source code the very same output of the tool can be used to refine the annotation process in an iterative procedure until a mature and solid annotation layer is obtained. The rest of this section aims to illustrate this process using the same systems described in the previous section, the "My Little Tomato Plant" system.

In order to proceed in an incremental fashion, the user might start focusing only in one particular architectural relationship. In this case, we assume the user selects the architectural relationship between the *Heat* applier component

and the *Gardener* Component, thus avoiding the need to completely annotate the code which might be an extremely demanding task. To accomplish this, all the components (excepting the *Heat* and *Gardener*) should be annotated as "Ignored". Running the tool under under this setting produces the following output shown in Figure 5:

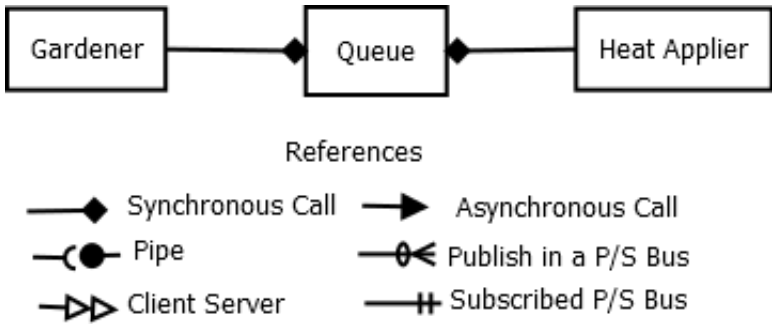

Fig. 5. Discovering the Pipe connector: Exploring the system without annotations

As it can be seen, the tool produces a view where the *Heat Component* communicates with a *Queue Component* with a *Synchronous Call* connector, and the same happens with the *Queue component* and the *Gardener Component*. Recall that any method call will be marked as a *Synchronous Call* since no annotations are needed for this type of connector.

Analyzing this figure the user might deduce that *Queue Component* is indeed the intermediate structure implementing the *Pipe connector* protocol and decides to annotate the *Queue* component's source code accordingly as show in the next code fragment (see Listing 7-10). Line 6 in Listing 7-10 exhibits the *PipePush* annotation attached to the method *push*, whereas line 10 in the same Listing shows the *PipePop* annotation attached to the method *pop* of the class *Queue*.

Listing 7-10. Component Queue source code

```
1 public class Queue {
2 private LinkedList<String> content;
3 public Queue(){
4 content = new LinkedList<String>();
5 }
6 @PipePush
7 public void push(String dato) {
8 content.push(dato);
9 }
10 @PipePop
11 public String pop(){
12 return content.pop();
```

```
13 }
14 public boolean hasElements(){
15 return !content.isEmpty();
16 }
17 }
```

Running the tool with the annotations shown in Listing 7-10 produces the following view depicted in Figure 6. It can be noted that the *Pipe* relationship between both components is now shown in the view.

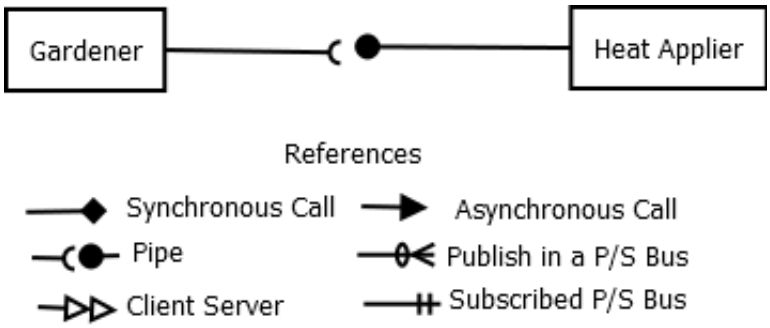

Fig. 6. Discovering the Pipe connector: Running the tool with an annotated source code

Next, the user might try to focus on the relationships between other components until the whole system in annotated. When this learning process is over the architectural view built by the tool can be compared against the architecture specification. It is worth mentioning that this process is only needed if there is little knowledge about the code implementing the system. It can be the case that the source code can be directly annotated without doing this step. It is only shown here as an example of the flexibility of the approach.

8 Case Study: Customized Connectors

In this section we describe how our tool was applied to capture a customized connector. Although the connector itself it is not included in our selected set of connectors (see Section 2), its behavior can be specified in terms of connectors in the set. This example shows that our tool does not only contemplate the mentioned connectors, but also any possible combination of them.

The case of study covers part of a *Twitter-based* application called *TW Rating*, which measures TV shows' ratings based on the comments made by the community in the *Twitter* social network. The application was developed by advanced students of Computer Engineering at the Universidad Nacional de Avellaneda in the context of a Software Engineering course. Regarding the size

of the analyzed system, it can be stated that the implementation of the connector consists of *11* Java classes, with a total of 1411 lines of code.

Twitter offers an application programming interface (API) that covers four main use cases [23, 13]:

- The user must authenticate in order to perform any action.
- The user can send a tweet to update its status.
- Users can retrieve tweets when they are online.
- Users can search for tweets using predefined topics called hashtags.

The developers were asked to implement a particular connector between the application and the *Twitter* API. This connector is presented in [23]. Instead of providing separate connectors for these four cases, work in [23] proposes a customized connector which internally handles all the required services. Three of them are based in a classic client-server interaction excepting the sending tweets service since tweets are asynchronously sent to *Twitter*. The others three, authentication, searching tweets and retrieving tweets, require a dedicated connection between the social network and the application. Figure 7 shows the architecture of the twitter connector. Reader is referred to [23] for a complete description of the behavior of the connector.

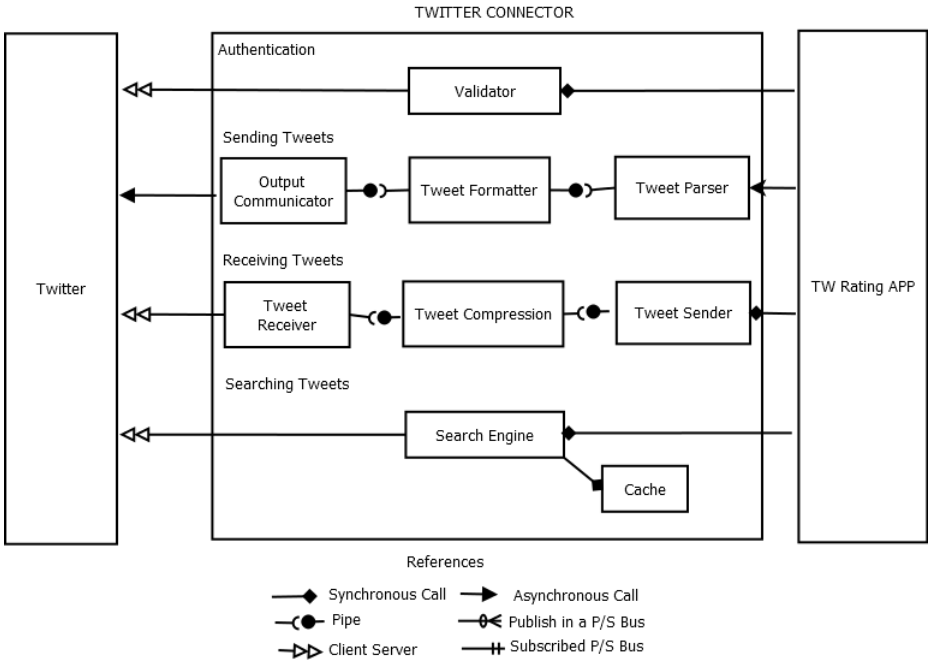

Fig. 7. Arquitecture Specification for the Twitter Connector

Using the localized and incremental method described in Section 7 the source code of the connector was isolated and incrementally annotated as well. Once this step was completed, the tool delivered the connectors view shown in Figure 8.

Analyzing the differences between the original architecture (Figure 7) and the one produced by the tool (Figure 8) it can be seen that two new components arise in the view built by the tool which are not present in the given specification of the connector. These components are: the *Pre Loader* component in the *Pipe and Filter* style implementing the *Sending Tweets* service and the *Pre Fetcher* component in the *Pipe and Filter* style implementing the *Retrieving Tweets* service.

When the developers were consulted about the gap between both architecture specifications they answered they needed to include these two new components due to changes made in the *Twitter API*. In a newer version of the API tweets can include videos and also contain multiple images instead of only one picture. The developers decided to include the mentioned new components to especially handle the multimedia content of this kind of tweets due to different purposes. For example, to avoid the degradation of the connection between *Twitter* and the application.

In this case, the tool was helpful to find an outdated architecture specification. The problem originated during the developing phase of the product, where a implementation decision was not properly documented. We also showed in this example how our tool can detect also customized connectors, taking into account the restriction that their behavior must be expressible as a combination of the chosen connectors.

9 Lessons Learned and Threats to Validity

In this section we discuss lessons learned while developing the case-studies. We also briefly compare our tool against others known tools and present some threats to validity related to our approach.

9.1 Lessons Learned

One of the first learned lesson is about the aspects implementation to detect the presence of some connectors. In order to detect the presence of the *Synchronous Call* connector we assume that any method call between two objects is treated as two objects communicating with a *Synchronous Call* connector. This implementation decision leads to the introduction of the "Ignored" annotation so that the user can indicate that certain methods calls should be ignored by the aspects during the execution of the system. This process might result in a tedious process. This could be avoided by introducing a dedicated annotation for the *Synchronous Call* connector and modifying accordingly its aspect definition. Under this context, the user only would only need to annotate those portions of the code he is interested in, and the "Ignored" annotation could be discarded.

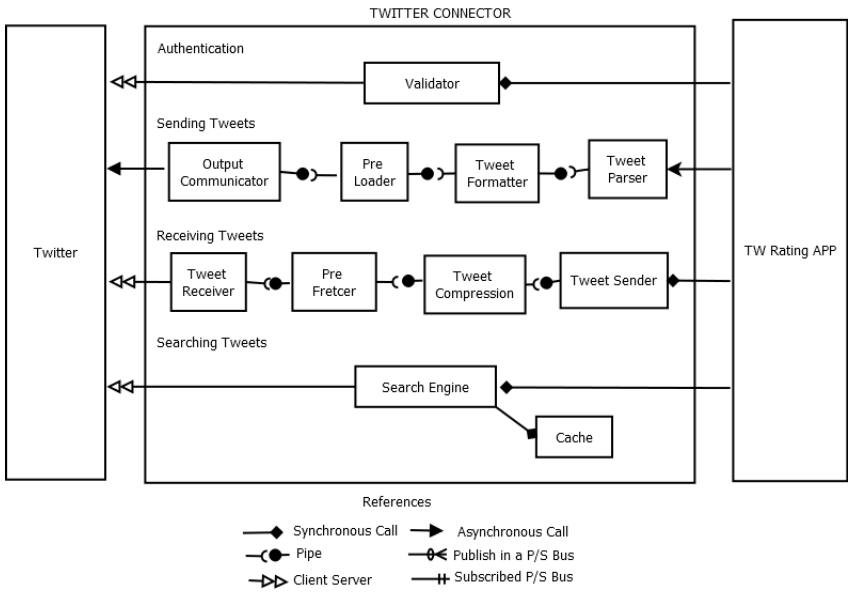

Fig. 8. Arquitecture View Built by the tool

We are considering to include this modification in the new version of our tool. Note that this modification does not invalidate the earlier described iterative process to discover and annotate the source code since the user can still learn from the obtained output of the tool to improve the annotation schema.

Similarly, current implementation details of others connectors deserve some considerations. The implementation of the *Asynchronous Call* rely on a particular coding convention, namely threads. After considering some other alternatives such as distributed environments we decided to use threads to maintain the simplicity of the tool in this first stage. Other implementation alternatives could be considered in future versions of the tool. More generally, given a proper aspect definition it would be possible to include simultaneously more than one aspect to detect several implementations for the same connector. An special consideration is to be analyzed in the case of using software libraries or frameworks to implement a given connector. Using our approach the user needs to annotate the portion of the code implementing the connector. If this code is not accessible to the user (because it is hidden in the library or framework) the connector could still be detected introducing some "ad-hoc" annotations. These annotations should indicate the place in the source code where the software library or framework is invoked.

Regarding the case study in section 8, a new connector is detected combining the behavior of the connectors included in our tool. These kind of "compound" connectors could be included in the set of connectors. This would require new aspects definitions and a more complex architectural analyzers infrastructure in

order to establish that, for example, a client-server connections is indeed part of a composite connector and not a "stand-alone" connector. All these points are certainly a challenge regarding future work. A possible initial step in this sense might be considering an object-oriented hierarchy structure based on abstractions and inheritance for the aspects definitions, similar to the hierarchy introduced in [12] when implementing object oriented patterns using aspect orientation.

Another important lesson came from comparing the output produced by our tool with other known approaches. Our tool can be seen as a lightweight tool since only detects the presence of connectors. Tools like [26, 21, 1] provide a more complex view including ports, roles, interfaces, component together with the possibility of performing automatic formal validation of the architecture behavior.

Similar to our work, [1] employs annotations in their approach. However, these annotations are based on local modular ownership in the code to impose a conceptual hierarchy on objects. Then, a static analysis extracts from the annotated program a global object graph that uses object hierarchy to convey architectural abstraction. Annotations in our work are directly related to architectural behavior, since they are strictly referring to the behavior of the connectors. In this sense, we believe the lightweight nature of our tool implies a simpler annotation effort. Nonetheless, a precise conclusion regarding this aspect requires a well designed experiment, which is out of the scope of this paper. Work in [26, 21] does not rely on annotations, but the source code must follow certain naming conventions, a condition that might hard to satisfy in certain contexts.

The current stage of our tool lacks the necessary infrastructure to provide more complex architectural reasoning such as the automatic detection of conformance violations or the presence of components. Components behavior in our approach are simplified since we can only detect objects communicating with each other. In order to properly detect components and more complex architectural behavior our tool would need to be extended beyond reasonings about method calls. This could be achieved by complementing our tool with other techniques such as introducing behavioral assertions like in [18] or employing static analysis as in [1].

9.2 Threats to validity

One of the main threats to validity relies on the size and complexity of our case studies. Although the analyzed systems present a rich and interesting architectural behavior they do belong into the small-sized systems category. Therefore, the usefulness of our tool need to be challenged with more complex and larger systems.

Another important topic is the expertise of the developers of the systems since in all cases students of Computer Science careers were involved. It might have been the case that the gap, errors and differences detected between architectural views in the case-studies (between the original architecture specification and

the one obtained by our tool) were originated due to the lack of experience of the developers. This indicate that the results obtained in this work should be validated against systems developed by more experienced developers.

Finally, the annotation process certainly represents a threat to validity due mainly of two reasons. In the first place, the systems were relatively small sized. And second, users annotating the source code were familiar to the systems. As it was previously stated a dedicated experiment is needed to effectively measure the amount of effort to annotate the source code.

10 Related Work

Approaches in related work can be divided into three categories [26, 21]. The first one groups those alternatives which aim to assure consistency between a system architecture's specification and its implementation by *construction* [2, 22]. These approaches work efficiently when the tools that give support to them can be actually employed [26]. 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 [19, 16]. These approaches aim to build the architecture of a system upon its code. However, they suffer from some known problems [26, 21]. 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 [1] uses static analysis and annotations to build a runtime architectural structure where conformance analysis can be applied. The main purpose of our work is different since we are only interested in building a dynamic view of the architecture. Similarly, annotations in [1] are focused in the structure and hierarchy of the system while in our work they are used to identify the behavior of the connectors. Work in [18] employs aspect orientation and annotations to verify conformance to architectural design. In this case annotations are used to introduce static assertions that trigger an error when non-valid interactions between modules take place. In our case, aspects orientation is dynamic since aspects monitors system's execution. Another difference is that our annotations are focused in the detection of connectors, and not to specify behavior of components. Nonetheless, our approach could incorporate these kind of annotations in order to detect the presence of components and validate their behavior.

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* [26, 21]. 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.

11 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 this can 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 non trivial case-studies 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 step will involve the need to rely on a more formal specification of the connectors behavior. We believe our current status being able to identify a potential use of the *Pipe and Filer* style, and the ability to pinpoint information between publishers and subscribers is promising enough to believe the expressivity of the tool can be easily increased. This next step would allow the possibility to interact with other software architecture tools like *Arch Java* [2] or *Disco-Tect* [26]. Similarly, we believe our approach can complement other alternatives based on hybrid approaches combining static and dynamic analysis such as [1]. We would also like to explore the possibility to annotate the code automatically. Finally, a next logical step is to validate our tool with more sophisticated examples including the possibility of interacting with software frameworks, which are increasingly present in actual software development.

References

1. 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.
2. J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *ICSE 2002*, pages 187–197. IEEE, 2002.
3. F. Asteasuain, C. Graiño, and D. Manuel. Dynamic validation of software architectural connectors. *ASSE 2016. JAIIIO. pp 87-98.*
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. J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *ACM Sigplan Notices*, volume 37, pages 161–173. ACM, 2002.
13. C. Honey and S. C. Herring. Beyond microblogging: Conversation and collaboration via twitter. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009.
14. https://www.redhat.com/es/files/resources/en-rhjb-fuse-eip-flashcards 10611447-.pdf. *Enterprise Integration Patterns*. Red Hat, 2015.
15. 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.
16. R. Kazman and S. J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *ASE*, 6(2):107–138, 1999.
17. 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.
18. P. Merson. Using aspect-oriented programming to enforce architecture, software engineering institute. Technical report, CMU/SEI-2007-TN-019, 2007.
19. 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.
20. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
21. B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *TSE*, 32(7):454–466, 2006.
22. 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.
23. T. Slotos. A specification schema for software connectors. In *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*, pages 139–148. ACM, 2014.
24. R. Suzuki. Interactive and collaborative source code annotation. In *ICSE*, pages 799–800. IEEE Press, 2015.
25. R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.
26. 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.