Temporal Superimposition of Aspects for Dynamic Software Architecture

Carlos E. Cuesta[1], María del Pilar Romay[2], Pablo de la Fuente[3], and Manuel Barrio-Solórzano[3] $\star$.

[1] Kybele, Departamento de Lenguajes y Sistemas Informáticos
ESCET, Universidad Rey Juan Carlos, Madrid (Spain)
`carlos.cuesta@urjc.es`
[2] Departamento de Sistemas Informáticos
Escuela Politécnica Superior, Universidad Europea de Madrid (Spain)
`pilar.romay@uem.es`
[3] Depto. de Informática (Arquitectura, C. Computación y Lenguajes)
E.T.S. Ingeniería Informática, Universidad de Valladolid (Spain)
`{pfuente,mbarrio}@infor.uva.es`

Abstract. The well-known Separation of Concerns Principle has been revisited by recent research, suggesting to go beyond the limits of traditional modularization. This has led to the definition of an orthogonal, *invasive* composition relationship, which can be used all along the software development process, taking several different forms. The object-like entity known as *aspect* is the best known among them, but in the most general case it can be defined as a new kind of structure. Software Architecture must be able to describe such a structure. Moreover, as most ADLs have a formal foundation, this can be used to provide an adequate formalization for the aspectual composition relationship, which is still under discussion. In this paper, we propose to base this architecture-level definition in the concept of *superimposition*, integrating the resulting framework into the process-algebraic, dynamic ADL named $\mathcal{P}i\mathcal{L}ar$. This language has a reflective design, which allows us to define that extension without redefining the semantics; in addition, the extended syntax can be used to avoid the use of reflective notions. Nevertheless, the language must provide the means to define general patterns to guide the weaving. Such patterns must not only identify locations in the architecture, but also the adequate states of the corresponding process structure. Therefore, we suggest to use *temporal logic*, specifically the μ-calculus, as the quantification mechanism. To illustrate this approach, we expose a case study in which all these ideas are used, and conclude by discussing how the combination of temporal logic and aspect superimposition, in this context, provides also an alternative way to describe architectural dynamism.

1 Introduction

From the very beginning, one of the basic guidelines of Software Engineering has been the *Separation of Concerns* Principle [20]. This is yet another translation of the classic strategy known as *divide et impera*, commonly attributed to Julius Caesar, to the

$\star$ This research has been partially financed by the Spanish Ministry of Education and Science under Projects MCYT-TIC2003-07804-C05-01 (DYNAMICA) and MCYT-TIC2003-09268.

computing field. The principle itself is at the core of every conceptual division within the Software Engineering body of knowledge, and its purpose is to separately deal with every detail in the development process, thus obtaining both simplicity and cohesion. It is also the deep reason behind the traditional practice of *modularization*, which causes the definition of structures within software; and also the motivation to create a specific discipline to study them, namely Software Architecture.

In recent years, this principle has been revisited and given birth to the approach known as Advanced Separation of Concerns, within which the so-called Aspect-Oriented Software Development [12] is the best known incarnation. Considered as a whole, it is basically an approach to software development in which those different *concerns* –or *aspects*– within a system are conceived and designed as separate entities. The result of this process is a set of overlapping functional elements or modules, maintainting mutual dependencies and crosscutting relationships. Traditional modular barriers are crossed; structural schemas, typically compositional and thus hierarchy-based, are no longer valid. An orthogonal, *invasive* composition [2] relationship is used instead.

This new kind of element, often known as *aspect*, was initially conceived within the boundaries of programming, at the implementation phase; but this origin has been superseded long ago. Nowadays, the principles of Aspect Orientation are applied all along the software lifecycle; in fact, there's even a specific term, namely *early aspects*, to refer to their influence in early stages of the development process, such as requirements specification and architectural design itself.

Therefore, to study the concept of aspect from an architectural point of view is not only reasonable, but even relevant. Moreover, as already exposed in [10], aspects are related to the notion of *architectural viewpoint* [24], still one of the more important and less studied in the field. However, existing Architecture Description Languages (ADLs) are conceived around the dimensions of composition and interaction, and designed to describe their structures. However, *aspectual* structures are not strictly compositional, as they have a different nature; so they are not easily specified using current ADLs. To adapt them to this sort of description, some specific *extension* must be defined.

In this paper we outline such an aspect-oriented extension for an existing ADL namely $\mathcal{P}i\mathcal{L}ar$ [9]. This language has a reflective basis, which has made us able to design this extension without affecting the semantics. But at the same time, the impact of the new syntax on the whole of the language has been greater than at first expected. On the one hand, the *aspectual* perspective can be used to avoid the complex reflective interpretation of the original; on the other hand, the new setting provides a whole new approach to describe *dynamic architecture*.

2 On the Notion of Superimposition

The concept of *superimposition*, also known as *superposition*, was first proposed in Concurrency Theory, both in the context of process algebras and action systems [3, 6]. It was originally conceived as a notion of refinement, relating different versions of a specification as variations from the original. The same approach was later used as an isomorphic notion of extension, composing a basic description with additional details. Consequently, superimposition is now conceived as a privileged relationship between two concurrent entities, such that the first one is able to access the internal details of the second one. This relationship has essentially a *compositional* nature.

Superimposition has also been approached from a different perspective, recently. In the search for an adequate formal foundation for the novel concepts identified in the context of Advanced Separation of Concerns, and particularly in Aspect Orientation, it has been regarded as a suitable candidate. The alternative composition offered by superimposition could possibly be assimilated to the *invasive* composition implicit in those concepts. In fact there are already several proposals relating this formal concept to the notion of aspect [14, 17, 21, 22], though most of them are located at a programming level. The only exception are Katara and Katz [15], who have also studied those concepts at the architectural level, but still using an informal approach.

There are several different definitions of superimposition in the literature. Though similar, they are not actually equivalent. Fiadeiro and Maibaum studied them from a categorical perspective [11], finding out that there are really three different flavours of the concept, which are respectively named *invasive*, *regulative* and *spectative* superimposition. The first one is the simplest as it is unrestricted; the third one is the most complex as it bears several restrictions to achieve better extensional properties.

In the architectural context, the most interesting among all the definitions of superimposition is probably Katz's [16], as it describes not a relationship, but an structure. The traditional strategy is to define the formal semantics for the superimposition relationship and then summarize it in a single compositional operator or morphism. Instead, Katz uses a different approach. He defines superimposition as a high-level concurrent *control structure* which implicitly uses the concept, and then provides the semantics for this construct. Therefore the relationship itself, a well-behaved form or spectative superimposition [11], is only indirectly defined.

This approach seems to be very adequate for the architectural domain, as the construct has a significant resemblance to certain presentations of the concept of connector, particularly higher-order connectors. Moreover, the resulting structure is also somewhat similar to *aspectual collaborations* [19], the most recent result of the work by Lieberherr *et al* on aspect orientation. This coincidence suggests that Katzian superimposition provides indeed a good starting point to explore aspectual composition at the architectural level, and therefore this is the definition to be used in the rest of this document.

2.1 Katzian Superimposition

In the following, the term *superimposition* will be generally used in the most general sense, but assuming Katz's definition [16] when necessary. Therefore, the expression *Katzian Superimposition* will be used to explicitly refer to this restricted meaning, and to concrete features in it which differ from some other approaches.

Structurally, every superimposition relationship has two parts: a superimposed element, and another(s) base element(s) where it is superimposed to. The complete structure receives the name of *superimposure* or *combination*. In the original conception, this is just *spectative superimposition*, in which the superimposed process refines or extends the original. It is able to inhibit the external interaction of the superimposee and also to *observe* its internal behaviour, but it cannot modify the latter [3, 16]. In summary, it doesn't have full control over it, and acts like a monitor.

The frequent use of such terms as *superimposed* or *superimposee* is rather confusing. To avoid complex periphrasis, we have decided to use a prefix-based notation, which states clearly the relative position of involved elements. Then, the superimposed component will be designated as *σ-component*, and it is conceived to be situated "over" some superimposee component, here known as *β-component*.

In Katzian superimposition, the relationship is defined as a structure which could simultaneously comprise several β-processes, such that the set of their σ-processes defines an algorithm –a behaviour– which is globally superimposed over a significant subsystem. The basic idea is that different σ-components may play different roles in this algorithm, thus providing us with the means to modularize the specification, while at the same time grouping these modules in a single construction.

Each one of those roles are defined as subprocesses in a Katzian superimposition, where they receive the name of *roletypes*. The same roletype could have several instances: that is to say, several elements could be playing the same role in the algorithm, and share the same description. In this case, several σ-processes, defined as the same roletype, are superimposed over several β-processes.

Thus the notion of *roletype* is very useful from an architectural perspective, and it would be used in the following sections. However, this could cause some confusion with the concept of *role* in a connector, which is somehow similar. To avoid this, we would use the name *role-component* to refer to the same notion in the architectural domain, as it has features in common to both ideas.

3 Aspect-Oriented Architecture in $\mathcal{P}i\mathcal{L}ar$

The $\mathcal{P}i\mathcal{L}ar$ [7–9] language is a dynamic, process-algebraic ADL, based on the notion of abstract process [8] and the concept of reflection, and with semantics founded on relation theory and the polymorphic π-calculus. The use of reflection is its distinguishing feature: as a consequence of that, a $\mathcal{P}i\mathcal{L}ar$ description could be stratified in multiple meta-levels, components are implicitly divided in three categories (base component, meta-component and meta-level component) and have a dual nature, and the definition of first-class connectors is not strictly necessary.

Our previous work [10] shows that the existing language, with the associated reflective support, was powerful enough to simulate the superimposition structure and the combination schema in an aspect-oriented architectural description. There, the foundation of our approach was the process-algebraic strategy of Andrews' definition [1]. But at the same time this approach was rather complex, and we suggested ourselves that an specific syntax for the new set of concepts would be rather convenient.

In the next sections we describe a proposal for an aspect-oriented extension of $\mathcal{P}i\mathcal{L}ar$, now using Katzian definition as a foundation. By doing so, we expose the real expressiveness of such an approach, and simultaneously provide a completely different perspective for the ADL, as the syntax acquires an alternative nature.

Of course this new vision does not exclude the previous one, though it can be used to *hide* it. Using it, we're able to describe the language without any mention of the concept of reflection or the meta-level hierarchy, but at the same time we retain most of the expressive power of the reflective vision.

New Concept	**Analogous Aspectual Notion**	**Former Reflective Concept**
Viewpoint	Concern	Reification Category
Architectural View	Crosscut	Meta-Level (subset)
Multi-dimensional Component	Hypermodule	(Extended) Metaspace
Architectural Fragment Partial Component	Aspectual Component Aspect, Hyperslice	Composite Meta-level Component Metaspace (subset)
Exterface	Aspect Interface	Metaface (Meta-Interface)
Bond Assertion	Pointcut Designator (Aspect) Connector	—
Superimposition (target)	Pointcut	Reification (target)
Role-Component	Pointcut (subject)	Reification (origin)
Superimposition (relationship)	Dynamic Weaving	Reification (relationship)
Combination (Superimposure)	Weaved *system*	Reification (set)
β-Component	Base *Module*	Base Component, Avatar
σ-Component	Aspect (part of)	Meta-Component, Rohatar
σ-Constraint	Advice	Meta-Constraint
Component in-a-Fragment	(Aspect) Wrapper	Meta-level Component, Niyatar
Bound β-Action	Join Point	Synchronization with Avatar

Table 1. Rough Conceptual Analogies in/to both $\mathcal{P}i\mathcal{L}ar$ Models

However, these *aspectual* and reflective perspectives of the same language are not conflicting at all; on the contrary, they naturally complement each other. So we're not rejecting the reflective interpretation, which is still more powerful; we're just providing an alternative explanation for the language, which allows us to initially avoid some of the most complex notions in the language.

3.1 PiLar Revisited: A New Vision For The Language

We have already exposed the reasons why we consider the description of *aspect-oriented* architectures to be relevant. An explicit aspect-oriented syntax is not strictly required, as we can use the reflective syntax to provide this description indirectly, as already exposed in [10]. However, this approach could allegedly be considered too complex. For this reason, we found it convenient to extend the language's syntax, such that relevant concepts can be directly managed.

This syntax extension would be based in Katzian superimposition, as this construct's shape provides an almost direct mapping to the architectural level. Specifically, the following concepts are introduced:

Architectural Fragment or **Partial Component**. This name designates the analogue of an aspect at the architectural level. Such an aspect is a new kind of module, similar to a component, but which was not designed to work on isolation; that's why it has a partial description. It is syntactically identical to a composite component, which unfolds as a Katzian superimposition.

Superimposition. Relationship which is implicitly introduced in the new model. It has the form of a Katzian construction, where the main structure is a fragment, the elements are plain components and role-components, instantiating as σ-components, and the resulting architecture defines a combination.

Role-Component. Each of the roles we can superimpose over a β-component when defining a Katzian structure. They are the "holes" in the architectural fragment, and they are filled by σ-components when the superimposition is made effective.

σ-Component. Every instance of a role-component, which is superimposed over a β-component. It can define also "external", non-superimposed behaviour.

β-Component. Every one of the base components where a σ-component is being superimposed, filling a gap in the fragment.

Combination. The set of all the elements involved in a Katzian superimposition, once it has been applied.

This version of the superimposition structure extends Katz's one merely by adding compositional details. Therefore, the fragment could have its own external interface, which is not projected into β-components; it could define its own constraints, which would be combined to those of its internal elements; and of course, a fragment definition can use additional components which are *not* going to be superimposed, that is, which would never act as role-components.

The new conceptual structure of the language is completely based on the implicit mapping between two structural relationships: the already existing, reflective notion of *reification*, and the concurrent concept of *superimposition*, introduced by the aspectual extension. Curiously enough, this idea is supported by the original semantics of the language, as there reflection is unfolded as a π-calculus structure of concurrent processes which is inspired [8] in another definition of superimposition [3].

The syntax is inspired in Katzian superimposition, and this means that this conceptual mapping would not be direct at the linguistic level. This means that some of the more complex aspectual notions are built over a set of several reflective elements; and also that some basic reflective concepts lack a peer in the aspectual view.

However in general terms, the mapping between the more important aspectual and reflective notions is rather intuitive. Every β-component is a base-component, and its σ-component is a meta-component. So, the notion of role-component is just a way to explicitly declare the meta-components in a fragment, and superimposition is just a reification relationship which is reflected over an already existing base component. Consequently, an architectural fragment is a composite meta-component, composed of one or several meta-components and (possibly) some additional meta-level components. Only the notion of combination lacks a reflective equivalent, as it combines elements which are situated in two different meta-levels.

There's no space here to provide a more detailed mapping between the reflective and aspectual concepts in $\mathcal{P}i\mathcal{L}ar$, as this is not our main concern here. Similarly, a detailed explanation of their similarities and differences which analogous notions in the specific field of Aspect Orientation would also require a rather long exposition, particularly to explain those analogues. This comparison is interesting anyway, so we provide a brief and compact summary both mappings in the Table 1.

3.2 A New Extended Syntax for the $\mathcal{P}i\mathcal{L}ar$ Language

Table 2 contains an enumeration of the new syntactic elements added to the language, which are based in the notions we have described in the previous section. Here we

Keyword	**Notion**	**Basic Structure**
$\backslash$**fragment**	Architectural Fragment	Analogous to a composite component including role-components.
rolecomp	Role-Component	Declaration of a component instance which acts as a σ-component. A “hole” in a fragment.
$\backslash$**exterface**	Exterface (External Interface)	Non-superimposed interface, reserved for the private use of the σ-component itself.
bcomp	β-Component	Prefix to designate elements of a β-component.
scomp	σ-Component	Prefix to designate non-superimposed elements.
impose	Superimposition	Dynamic operator to superimpose a fragment over several β-components (see Figure 4).
del	Unweaving	Deletion (destruction) of a superimposition.
$\backslash$**bond assertion**	Bond Asssertion	Syntax to select relevant join points.

Table 2. Aspect-Oriented Extension for $\mathcal{P}i\mathcal{L}ar$: Syntax

will not try to describe the minor details of this syntax, as they are rather intuitive, and anyway most of them will be used later for the case study included in section 4.

The only notion we have not mentioned before is that of *bond assertion*. This is the incarnation of the quantification mechanism which is necessary in every aspect-oriented language [13, 10]. As we expose in the next section, this mechanism is based on *temporal logic*, and it provides the syntax to select join points at any place *or moment* in the architecture, resulting in a truly dynamic weaving mechanism.

Of course, already existing introspective (reflective) operands in the language can still be used. In fact, some of them are even essential to outline a good aspect-oriented description, as they fill the role of so-called *aspectual reflection* [18] abstractions. For example, the language already included a reflective operator, **bound**, to obtain the set of links bound to a given port (or the set of ports pount by a given link). This operator happens to be also particularly useful in an aspectual context.

Most of this “extended” syntax is actually just *syntactic sugar* for aspect-oriented abstractions. Not a single element in the language semantics has required to be adapted to the new conception of the ADL. This is in accordance to our initial purpose, in which this new version of $\mathcal{P}i\mathcal{L}ar$ is conceived just as a different presentation of the same language, which tries to avoid the use of reflective notions.

However there is an exception to this rule: the definition of assertions and the use of temporal logic *is* actually a new addition to the language. But this addition has not been an arbitrary decision; as explained next, there are several reasons to use it.

3.3 The Syntax of Temporal Assertions

The existence of some quantification mechanism is strictly necessary for a sensible Aspect Orientation definition [13]. Without it, every join point between two structures has to be individually designated. Though this would still be useful [10], it is not flexible enough; an architectural description is supposed to describe structural *patterns*, and therefore the lack of a general expression to refer to patterns of superimposition would be considered as a severe limitation.

Moreover, in the context of dynamic architectures, the choice of a particular join point to superimpose an architectural aspect does not only depends on the system's structure, but frequently also on the concrete *situation* or state in which an element (or set of elements) is. Superimposition happens not only at a *place* in the architecture, but also at a particular *moment* in the system's evolution.

Existing aspect-oriented languages, at the programming level, use quantification mechanisms based just on name structure; this would be a very inadequate approach at the architectural level. Some other proposals provide a better mechanism by suggesting the use of some variant of predicate logic. While this is much more flexible, it is still not enough, particularly in the presence of time. Besides, classical logic is probably not the best choice to combine with the semantics of a process-algebraic ADL, which would usually consist of transition systems.

Then our proposal tries to be a solution to both problems, and it is based on the addition of *temporal logic* to the language, using the form of assertions or laws. In the context of $\mathcal{P}i\mathcal{L}ar$, which is founded on a process algebra and the notion of bisimulation, the obvious choice is the modal μ-calculus [4], a branching-time temporal logic, which is also considered as the most general among them.

Therefore, the syntax for temporal assertions would be based in that of $\mathcal{P}i\mathcal{L}ar$'s dynamic language and the μ-calculus. There are several different but equivalent notations for the latter; here we follow Stirling's [4, 23], probably the best known among them. Currently we would only use the basic syntax; but this is just a first approach to the problem, so it should not be taken as a final decision. We could consider further additions, like pure temporal operators in the CTL style, which are usually considered easier to understand by the average software engineer. Those would be syntactic sugar anyway, as their semantics are already expressible in the μ-calculus syntax.

The basic extension is just a notion of *law* or *assertion*. With this addition, the language acquires a new quality, as it gets transformed into some sort of *Law-Governed* $\mathcal{P}i\mathcal{L}ar$, which is even capable of describing architectural styles. However, subject to this notion there's a set of new concepts, which are summarized in the following.

Assertion. Following Lamport, we use the term *assertion* to refer to any temporal formula defined over the architecture. The purpose of this term is to easily separate them from behavioural constraints, which in $\mathcal{P}i\mathcal{L}ar$ are defined as processes. The **assertion** structure of the syntax is defined to contain these formulae.

Bond Assertion. The only difference between this and a conventional assertion is that this is *active*. This means that when the formula requires an action to happen, and this action is not observed, the assertion itself is in charge of doing it, *but only if this action is related to a superimposition*. Expressed otherwise, if an assertion states that an **impose** action must happen, the assertion itself is the one which creates a superimposition. Apart from being prefixed with a **bond** qualifier, the syntax is identical to that of a conventional assertion.

Action. Any valid action in a $\mathcal{P}i\mathcal{L}ar$ specification, in particular message inputs and outputs through some port. They are the set of observable events from the assertion's point of view. The syntax allows to specify a single action or an enumeration of several ones. When it is prefixed by a minus (–) sign, this refers to the set of each action *except for* this. Conversely, the asterisk ($*$) refers to every of them.

Bound Name. This is not a μ-calculus notion, as it comes from aspect orientation. As noted above, assertions are used as a quantification mechanism, and they observe actions happening in *any port* of the namespace. Then such a port is a join point within a component, and thus we would often need to refer to it again. To be able to do that, we provide a mechanism to *bind* the name of these elements within a special variable defined for this purpose. The binding process must comply with the Scope Inversion Rule, stated below.

Possibility Modality. When referencing an action, this means that if the action happens, the expression which follows in the assertion *may* be true. The purpose is to state that the system is able to do something in this point of its evolution. It is expressed by enclosing the action $\langle a \rangle$ in angles.

Necessity Modality. The second alternative. When referencing an action, this means that if it happens, the expression which follows *must* be true. The purpose is to *forbid any other* possibility to happen in the system, indicating an inhibition. It is expressed by enclosing the action $[a]$ in brackets.

Minimal Fixpoint. It has a complex semantics; but we can summarize it [4] by saying that it specifies a repetition of undefined, but *finite* length. In the μ-calculus, it is often expressed as μ (or min); in $\mathcal{P}i\mathcal{L}ar$, we shall use the keyword **nrec**.

Maximal Fixpoint. It is equally complex; we can summarize it by saying that it specifies a repetition of *infinite* length. In the μ-calculus, it is often expressed as ν (or max); in $\mathcal{P}i\mathcal{L}ar$, we shall use the keyword **xrec**.

This construction has the same semantics as the equivalent notions in the μ-calculus, and therefore it has been already formally defined [4, 23]. There's only one difference from their usual application to a process algebra: here we don't have a flat namespace, but a hierarchy of names. This results in two consequences. First, any assertion must be defined over a concrete namespace, provided by a component; this is designated by using the **over** clause. Second, it is often necessary to bind the name of the components in which actions are observed. As stated above, this binding process must comply with the rule which follows:

Rule 1 (Scope Inversion) *Every action on a port which is observed in an assertion binds the name of the* innermost *component where this port belongs in the composition hierarchy, using the conventional syntax to qualify those names.*

Next we will expose an example to show how an assertion works. To ease the explanation, and also to show the differences in the notation, we would use the same one which is later provided in $\mathcal{P}i\mathcal{L}ar$ syntax in Figure 4. Moreover, the actions to be observed (acc_1 and acc_2) have been abstracted, so that we just focus on the temporal aspects of the formulae and not on the concrete behaviour.

This assertion[4] uses the conventional notation of the μ-calculus [4]. The mapping to the syntax in $\mathcal{P}i\mathcal{L}ar$ should be apparent by comparison to the descripiton in the Figure 4, taking into account that the actions (acc_1 and acc_2) are themselves $\mathcal{P}i\mathcal{L}ar$ actions, expressed in the syntax of the dynamic language.

[4] This is of course just a single assertion (*Always_Do_Tunnel*), but it has been divided in three parts to ease its explanation. The specification in a single formula can use a much more compact notation, which is: $\nu X. \, ([acc_1] \, ([-acc_2]\mathbf{false} \, \wedge \, \langle - \rangle \mathbf{true}) \, \wedge \, [-] \, X)$.

$$\text{Must\_Tunnel} ::= [-acc_2]\,\mathbf{false} \wedge \langle - \rangle \mathbf{true} \tag{1}$$

$$\text{Do\_Tunnel} ::= [acc_1]\,\text{Must\_Tunnel} \tag{2}$$

$$\text{Always\_Do\_Tunnel} ::= \nu\, X.\, (\text{Do\_Tunnel} \wedge [-]\, X) \tag{3}$$

The assertion has been separated in three formulae, such that each one of them is contained in the following. So we begin with the most internal one (1). It describes the conjuction of two terms: the second states that it's possible for any action to happen; the first states that when something happens which is *not* the acc_2 action, the formula gets false. This means that the conjunction only gets true if the acc_2 action actually happens. This is the μ-calculus way to indicate that something is mandatory.

The second equation (2) is trivial; it just states that after an acc_1 action happens, the previous one (1) is *necessarily* true; in summary, acc_2 must happen.

The last equation (3) encloses the former one in another conjunction, inside the scope of a maximal fixpoint ($\nu\ X$). The other part of the conjunction is enabling any action to happen, assuming that the fixpoint X (that is, any possible future) gets true. For this to be consistent, if the acc_1 action happens at some point in time, the other equation (2) must be true, and so this forces us to "trigger" acc_2. This is a maximal fixpoint, therefore this sequence can happen as many times as required ("always").

In summary: the assertion states that an acc_1 action may happen at any moment, but in the case the next action is necessarily acc_2; and this is always true, meaning that this happens *every time* the action acc_1 is observed.

4 Case Study: P/S Architecture with Secure C/S Connection

In this section we provide a case study outlining the use of the new aspectual framework, to show how it can be applied for the purposes of architectural description. To simplify things, we use an augmentative (asymmetric) model, which *grows* from an initial basis; a compositive model, though symmetric, would have been much more complex. The general idea is that we begin with a base architecture, and then we superimpose a *security aspect* over it, defining the weaving as a bond assertion. Besides, this superimposition indirectly modifies the system's structure; therefore, this is also an example of a new way to describe *architectural dynamism*, a very interesting side-effect of the aspectual framework.

The case study describes a hybrid architecture, blending the Publisher/Subscriber[5] and the Client/Server architectural styles [5]. The global conception is that of a distributed system composed of a number of *subscribers* which contract the services of a *publisher*; for instance, a news service. As soon as new contents are made available, the publisher notifies subscribers by triggering an event. When a subscriber observes this event, it must decide whether it is interested in those contents or not. If this is the case, the subscriber starts to behave like a *client*, which tries to communicate to a *server*; but the connection to this server has yet to be created. In this particular moment, a *secure*

[5] This architectural pattern has also been described in the literature as the *Implicit Invocation* architectural style, and even the *Observer* design pattern. The structure is fairly identical.

```
\component Publisher (
  \interface ( port notify | port server )
  \constraint (
     Provide def= ( Publish | Serve )

     Publish def= rep ( tau (msg);
              loopSet ( bound(notify) )
                       ( notify !(msg) ) )

     Serve def= rep ( server?(req);
             tau (req,data); server!(data ) ) ) )

\component Subscriber (
  \interface (
     port receive | port client )
  \constraint (
     Observe def= rep ( receive?(msg);
             tau (msg,ask,req);
             if (ask) ( Request(req) ) )

     Request(req) def= ( client !(req);
             client ?(data); tau (data ) ) ) )

\component System (
  \config (
     PS: Publisher | S1, S2, S3: Subscriber |
     \bind (
        PS.notify = S1.receive |
        PS.notify = S2.receive |
        PS.notify = S3.receive ) ) )
```

Fig. 1. Hybrid Publish/Subscribe and Client/Server Architecture (w/o connection)

connection among them is created, applying a *tunneling* protocol which ensures that every interaction between them is encrypted in advance. Using this connection, the client receives the selected contents. This process may happen as many times as required.

Many details of this example which are not strictly related to this paper's subject have been left out, as we try to briefly expose an averagely complex system. Therefore, there are details on the final system which don't try to be realistic. Then, the specification shows how new private connections are created, but they are never destroyed; the only reason to omit this step is to keep the example short and simple enough.

The case study is described in Figures 1 to 4. The first one provides the base Publish/Subscribe architecture, and also the Client/Server infrastructure. The second one describes an architectural fragment, which provides the secure connection by using the tunneling protocol. In the last one we provide the bond assertion, which dynamically "triggers" the superimposition of this fragment to the base architecture.

The specification in Figure 1 is therefore fairly standard. The *Publisher* component has two ports, enabling it to act as publisher or server, as required; and two constraints, *Publish* and *Serve*, which control any interaction in these ports. The first process starts when some new content –expressed as an internal **tau** action– is created; then this is notified to every subscriber connected to the *notify* port. The second process describes how the server waits to receive some request; when this happens, it locates the requested data, and sends them to the requesting client.

On the other hand, the *Subscriber* component can similarly behave either as a subscriber or a client. However in this case the two relevant constraints are related. The first process, *Observe*, waits to receive a notification event, which is internally evaluated. If an affirmative decision is taken, the component begins to behave as a client by starting the *Request* process. This just sends a request for the new content, waits to receive the result, and then processes the information.

The *System* component is just a composite to define the whole of the system. Let us note that initially, only the publisher and its subscribers are connected.

Figure 2 describes an architectural fragment defining the superimposition of a secure connection, supported by a tunneling protocol, over the previous base architecture. So it is the equivalent of a *security aspect*. As this is conceived in an augmentative model, a high degree of connascence with the β-architecture is allowed, something that could be less adequate in a symmetric model.

```
\component TBegin (
  \interface ( port send )
  \constraint (
      lock ( bcomp.client );
      rep  ( catch client(req ); send!(req);
             send?(ex); tau (ex,x);
             shift  client (x ) ) ) )

\component TEnd (
\interface (
      port recv )
```

```
  \constraint (
      lock ( bcomp.server );
      rep  ( recv?(req ); shift server(req);
             catch server(x);
             tau (x,ex ); recv!(ex ) ) ) )

\fragment Tunnel (
  \config (
    rolecomp TB: TBegin |
    rolecomp TE: TEnd |
    \bind ( TS.send = TR.recv ) ) )
```

Fig. 2. Architectural Fragment to Superimpose a Secure Connection

The *Tunnel* fragment is defined as a Katzian superimposition, built as the composition of two role-components connected by a basic link. These components define a *tunnel* using this link: every message to be sent is encrypted in advance on origin, and only the legitimate receiver would know how to decrypt it. This way a *secure channel* is created over a conventional connection.

The tunnel has been designed to be asymmetric: only the sending of data is encrypted, while the requests are not, as they are not considered as sensible information. This implies that interaction is always initiated by the same part of the interaction. As a result, the tunnel is conceived as having an explicit beginning and an end, as indicated by the archtypes *TBegin* and *TEnd*, which respectively describe the behaviour to be superimposed over every client and the server.

The former has then been designed to be combined with a *Subscriber*. First, it locks the *client* β-port, thus inhibiting any further uncontrolled interaction. Then it captures (*catch*) any message sent through this port, which must be a request. This request is sent to the server unaltered, using the superimposed connection. Eventually, some *encrypted* response is received, and it must be decrypted; this is made by an internal process (*tau*). The requested data are then obtained, and then they are inserted (*shift*) into the β-port. All this process is transparent to the oblivious client; it just requests some data, and later receives those data in the same port.

The *TEnd* archtype is similar, but it gets superimposed to a *Publisher*. Equivalently, it locks the β-port *server*, but now it waits for a request on the superimposed connection. When this is received, it is inserted into the server component, which “believes” this to be a conventional reception, and answers by providing the relevant data. Those data are captured by the σ-component, which encrypts them in an internal process, and sends the result through the σ-connection.

Figure 3 depicts and summarizes the architecture of the augmented system. Two long arrows represent the superimposition relationship, which imposes *TBegin* and *TEnd* to *Subscriber* and *Publisher*, respectively. Small arrows represent the flow of information within the woven architecture.

Finally, Figure 4 describes the bond assertion which blends the fragment with the β-architecture. Once we’re aware of the meaning of the temporal formula, which was described in section 3.3, the explanation is immediate.

First, the assertion is declared: it is a bond assertion, defined over the namespace of the *System* component, and the name of the main formula is *Always_Do_Tunnel*; the rest are subformulae. The temporal expression is now evaluated; it means that any

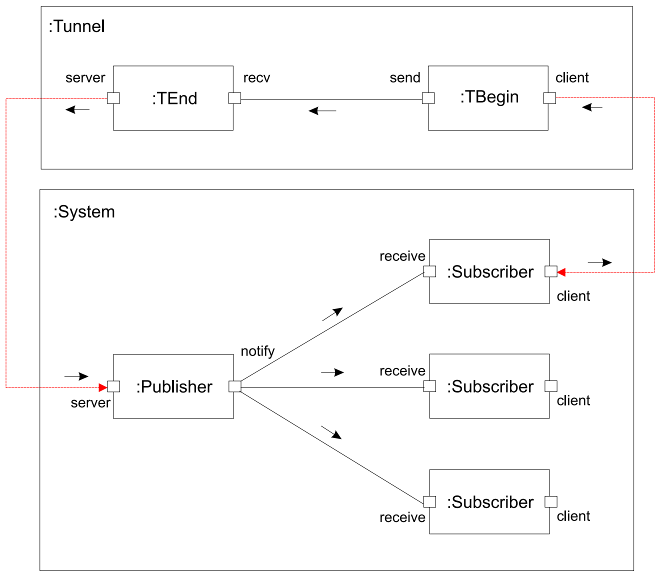

Fig. 3. Publisher/Subscriber System with a superimposed Client/Server Tunnel

```
\bond assertion Always_Do_Tunnel  over System (
    Must_Tunnel(s)        is=   [ – impose Tunnel (s | PS ) ] false  and <∗> true
    Do_Tunnel             is=   (name Sub) [ Sub.client!(_) ] Must_Tunnel(Sub)
    Always_Do_Tunnel      is=   xrec X ( Do_Tunnel  and [∗] X ) )
```

Fig. 4. Bond Assertion: Superimposing the *Tunneling* Aspect

action may indefinitely (*xrec*) happen; but if this action is some kind of sending through a port named *client*, the name of the sending component is bound in a *Sub* variable; data themselves are ignored. Now the next step is mandatory, as the formula requires: the *Tunnel* fragment is superimposed over the bounded client and the standalone server. This is a bond assertion, so it is the one which creates the superimposition; this way, the fragment is *woven* into the architecture.

In summary, every time (*always*) a client sends a data request, a *secure connection* between it and the server is transparently created.

Though simple, this example has yet another reason to be notorious; it shows how the dynamic weaving (combination) of architectural aspects, based on the temporal logic support, causes in fact a *dynamic evolution* of the architecture. Temporal logic has been used before in the context of Architecture, but always for analyzing purposes, never to describe a system, or a dynamic effect within it. As far as we know, this is the first time that it is used to cause an effect on the system's structure.

5 Conclusions and Future Work

An ADL must be first conceived as a formal description language. This means that it is used to describe the structure and properties of an architecture, and this can be done just for specification purposes. From this point of view, the existing temporal logic support is adequate for our purposes, as long as it can be used to capture the system's behaviour. But, an ADL can also be used to simulate (or even generate) the system itself; in that case, the existing temporal support would not be enough. Some kind of *timeout* mechanism would be required to limit the timeframe and ensure that a particular superimposition indeed happens; a minimal fixpoint ("eventually") formula would not suffice to provide the required behaviour.

This paper introduces superimposition as "the" third architectural dimension, a role which has previously been played by reflection. This is orthogonal to the traditional dimensions of composition and interaction. The expressiveness of this approach is provided by combining elements located at different places in those three dimensions, even indirectly. Though not as expressive as reflection, it provides still a very high degree of flexibility, and a new approach to tackle the problem of architectural dynamism.

However, this implies *aspect composition*, something that at present has only been tackled by using heuristic techniques. The only notorious exception is a work by Sihman [21, 22], which is also based on Katzian superimposition; so their results could be considered in the context of $\mathcal{P}i\mathcal{L}ar$. A detailed study is scheduled as future work. This work will also include several related questions, such as the management of priorities and dependencies between aspects. These problems are still open questions in research within Aspect Orientation, and therefore any results at the architectural level would be of general interest to the whole field.

On the other hand, the introduction of assertions in the language has been as generic as possible, as it does not only provide the support to bind and weave aspects, but also to define architectural styles. This is obviously a feature we have not exploited in this paper; though very promising, it has yet to be carefully evaluated.

In summary, the introduction of the notion of "aspects" in Software Architecture does not only provide the means to describe several new abstractions, but at the same time simplifies the presentation of previous approaches, and outlines a whole new range of applications, namely the specification of dynamism, the definition of multiple architectural views, the separate description of concrete concerns, such as security or coordination, or the study of new composition schemes.

References

1. James H. Andrews. Process-Algebraic Foundations of Aspect-Oriented Programming. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Reflection 2001: Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, September 2001.
2. Uwe Aßmann. *Invasive Software Composition*. Springer Verlag, 2003.
3. Luc Bougé and Nissim Francez. A Compositional Approach to Superimposition. In *15th Annual ACM Symposium on Principles of Programming Languages, POPL'88*, pages 240–249, San Diego, 1988. ACM Press.
4. Julian C. Bradfield and Colin P. Stirling. Modal Logics and mu-Calculi: An Introduction. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier Science B.V., 2001.
5. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

6. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
7. Carlos E. Cuesta. *Reflection-based Dynamic Software Architecture*. ProQuest Information & Learning, Madrid, May 2003.
8. Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, and Encarnación Beato. An "Abstract Process" Approach to Algebraic Dynamic Architecture Description. *Journal of Logic and Algebraic Programming*, 63(2):177–214, May 2005.
9. Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio Solórzano, and M. Encarnación Beato. Introducing Reflection in Architecture Description Languages. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Software Architecture: System Design, Development and Maintenance*, chapter 9, pages 143–156. Kluwer, August 2002.
10. Carlos E. Cuesta, M. Pilar Romay, Pablo de la Fuente, and Manuel Barrio-Solórzano. Reflection-based, Aspect-oriented Software Architecture. In Flavio Oquendo, Brian Warboys, and Ron Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 43–56, May 2004.
11. José Luiz Fiadeiro and Tom S.E. Maibaum. Categorical Semantics of Parallel Program Design. *Science of Computer Programming*, 28(2–3):111–138, 1997.
12. Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors. *Aspect-Oriented Software Development*. The Object Technology Series. Addison-Wesley, October 2004.
13. Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns (ASoC'2000)*, October 2000.
14. Mika Katara. Superposing UML class diagrams. In *AOSD'02 First Workshop on Aspect-Oriented Modeling with UML (AOM1)*, Enschede, The Netherlands, April 2002.
15. Mika Katara and Shmuel Katz. Architectural Views of Aspects. In *Proceedings of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 1–10. ACM Press, March 2003.
16. Shmuel Katz. A Superimposition Control Construct for Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
17. Pertti Kellomäki. A Formal Basis for Aspect-Oriented Specification with Superposition. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages*, pages 27–32, April 2002. ISU-TR02-06.
18. Sergei Kojarski, Karl Lieberherr, David H. Lorenz, and Robert Hirschfeld. Aspectual Reflection. In *Proceedings of SPLAT'03*, March 2003.
19. Karl Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, September 2003.
20. David Lorge Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
21. Marcelo Sihman and Shmuel Katz. A Calculus of Superimpositions for Distributed Systems. In *Proceedings of the First International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 28–40. ACM Press, April 2002.
22. Marcelo Sihman and Shmuel Katz. Superimpositions and Aspect-Oriented Programming. *The Computer Journal*, 46(5):529–541, September 2003.
23. Colin Stirling. Modal and Temporal Logics. In Samson Abramsky, Dov Gabbay, and Tom S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1991.
24. Eóin Woods. Experiences Using Viewpoints for Information Systems Architecture: an Industrial Experience Report. In Flavio Oquendo, Brian Warboys, and Ron Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 182–193, St. Andrews, UK, May 2004. Springer Verlag.