Architectural Description of Object Oriented Frameworks: an Approach

Gabriela Arevalo[*,**] **— Isabelle Borne**[**]

*LIFIA, Facultad de Informatica, Universidad Nacional de La Plata, La Plata, Argentina
** Ecole des Mines de Nantes, 4 rue Alfred Kastler, B.P. 20722, Nantes Cedex 3, France
garevalo@eleve.emn.fr, borne@emn.fr

ABSTRACT. Integration of architectural descriptions in development tools and environments, in order to take architectural descriptions into account, is a topical issue. Nowadays, the existing formalisms to represent software architecture fail in providing a clear semantics and only give an intuitive graphical representation of the system as a whole. More specifically, the framework architectures should show the overall design and the specification of the points of the variability of the framework, making easier the reuse of the architectures, integration with others frameworks and a reference to measure the changes in subsequent versions of the frameworks. In this paper we propose an approach to describe the architecture of frameworks, combining formal and non-formal formalisms: Wright, an architectural description language developed at Carnegie Mellon University, and architectural patterns. Based on the study of several frameworks, our objective was to produce a complete description of a framework, to show the expressive power of both approaches and to consider complementarity and flexibility regarding to other approaches.

RÉSUMÉ. L'intégration de descriptions architecturales dans des outils et des environnements de développement, prenant en compte l'intégration de ces architectures est un sujet d'actualité. Les formalismes existants de représentation des architecture ne fournissent pas une sémantique claire et donnent seulement une représentation graphique intuitive du système. Plus spécifiquement, les architectures de framework doivent montrer la conception dans son ensemble et la spécification des points de variations du framework, facilitant la réutilisation des architectures, l'intégration avec d'autres frameworks et une référence pour mesurer les changements de versions du framework. Dans cet article, nous proposons une approche pour décrire l'architecture de frameworks, combinant des formalismes formels et informels : le langage de description d'architecture Wright, développé à l'université de Carnegie Mellon, et les patterns d'architecture. En s'appuyant sur l'étude de plusieurs frameworks, notre objectif fut de produire une description complète d'un framework, de

montrer le pouvoir expressif des deux approches et de considérer leur complémentarité et flexibilité par rapport à d'autres approches possibles.

KEY WORDS: *software architectures, object-oriented frameworks, Architecture Description Languages.*

MOTS-CLÉS : *architectures logicielles, framework objet, langages de description d'architecture.*

---

1. Introduction

Software Architecture is an area of study within the software engineering community for already quite a long time, that has become a topic of interest within the object-oriented community as well. Integration of architectural descriptions in development tools and environments, in order to take architectural descriptions into account, is a topical issue. One approach consists in starting by architectural styles' descriptions, that allow to underline families of systems for which frameworks can be easily deduced. An object-oriented framework is a kind of reusable software architecture comprising both design and code. More specifically, [Mat96] defines an object-oriented framework as "a generative architecture for maximum reuse represented as a collective set of abstract and concrete classes, encapsulating potential behaviour for subclassed specialisations". The first objective of this work was to study architectural descriptions that can be used for object-oriented component frameworks (collections of software components), particularly the ones based on architectural patterns and composition languages

Description and documentation are closely related. One critical issue for users and implementors of a framework is the documentation that explains what the framework provides and what is required to instantiate it correctly for some application. Typically, a framework is specified using a combination of informal and semi-formal documentation. On the informal side are guidelines and high-level descriptions of usage scenarios, tips and examples. If an object-oriented methodology, such as UML [UML97], is used to document the framework, there are class and collaboration diagrams as a description artefacts. These approaches tend to be informal and idiosyncratic, consisting of box-and-line diagrams that convey the essential system structure, together with the prose that explains the meaning of the symbols [MKMG96]. On the semi-formal side one usually finds a description of an application programmer's interface (API) that explains what kinds of services are provided by the framework. APIs are formal to the extent that they provide precise descriptions of those services -usually as a set of signatures, possibly annotated with informal pre and post-conditions [SG99].

This documentation is clearly necessary, but it leaves many important questions unanswered for component developers, system integrators, and framework implementors. For example, the framework API may specify the names and parameters of services provided by the infrastructure. However, it may not be clear what are the restrictions (if any) on the ordering of invocations of those services. Usage scenarios may help, but they only provide examples of selected interactions,

requiring the reader to infer the general rule. Moreover, it may not be clear what facilities must be provided by the parts added to the framework, and which are optional. [SG99]. As with most forms of informal system documentation and specification, the situation could be greatly improved if one had a precise description as a formal specification of the framework.

There are several reasons to have an architectural description of a framework based on high-level interfaces and interactions, and characterising their semantics in terms of protocols:

- *Reuse of architecture*: Transmitting a language-independent view of the architecture allows the high-level design of the framework to be reused in implementing it in other languages, or in modifying it for use in other domains. [Ric98]
- *Integration of frameworks*: In order to facilitate the construction of systems from several existing frameworks, the architectural assumptions of each framework should be made explicit. [GAO95] [MB00]
- *Evolution and re-engineering*: Having an architectural description of a framework gives us a reference against which one can measure the changes in subsequent versions of the framework. In the same way, the ability to describe the architecture of an application allows us to form hypothesis about the architecture which can be tested in the process of reverse engineering [MN95].

In this paper we propose an approach to describe the architecture of frameworks, combining formal and non-formal formalisms: Wright, an architectural description language developed at the University of Carnegie Mellon, and architectural patterns. The complete description of this work can be found in [Are00], here we will focus on the mapping between source code (written in Smalltalk and Java) and CSP process, since the formal basis of Wright is CSP [Hoa85], and on the methodology proposed to get the architectural description of a framework.

2. Related Work

The particular combination in the use of formal languages to describe an object-oriented framework is only shown in [SG99]. In this work, they develop a specification of Sun's Enterprise Java-Beans. Firstly, they show formal architectural models based on protocols clarifying the intent of an integration framework, as well as exposing critical properties of it. Secondly, they describe techniques to create the model, and structure it to support traceability, tractability, and automated analysis. This work is a good approximation on ways to provide formal architectural models of object oriented frameworks.

Recent volumes on application frameworks [FSJ 99] relates some studies on framework documentation and description, but not from an architectural point of view.

3. Architectural Description of a Framework

3.1. *Software Architecture and object-oriented framework*

There are many valid definitions of software architecture, we choose the one considering software architecture as the structure of the components of a system, their interrelationships, and principles and guidelines governing their design and evolution over time. In our study we address more specifically the description of object-oriented frameworks, that are reusable architectures of systems that describe how the system is decomposed into a set of interacting objects. We have also restricted our work to how classes are related structurally in a framework and what the consequences of this structure are on the software system.

3.2. *Software Components = Architectural Components ?*

At the programming language level, components may be represented as classes of objects or a set of classes, since the common requirements to be a software component are fulfilled by an object: encapsulated information, specified interface, context dependencies and used as a building block. Thus, we can provide a direct mapping between a software component (e.g. a class) and an architectural component. This process is transparent in the developed mapping. The interface of the architectural components will be the services provided by the classes.

3.3. *Connectors*

Components do not represent anything by themselves, and the most relevant part is the possibility of connecting them with different types of relationships (defined as connectors in [SG99]). Defining the architectural structure of a framework, another key question is what are the connectors. It is essential to have a clear distinction between the classes, and the mechanisms that co-ordinate their interaction. In this way, we isolate two models in the architecture: one for the communication and one for the computation. Firstly, Let us see two ways of identifying possible connectors presented in [SG99].

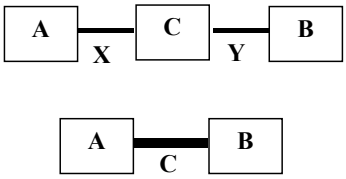

Figure 1: Component or Connector ? [SG99]

Consider a system consisting of three components : A, B, and C (figure 1). In some cases the purpose of C is to enable the communication between A and B, using A-C specific protocol over connector X, and C-B protocol over connector Y. On the one hand, i f those two protocols are completely independent, it makes sense to represent C as a distinct component, and keep X and Y as separate connectors. On the other hand, if events on X are tightly coupled with those on Y (or vice versa), then, it makes more sense to represent the protocol between X and Y directly using a single connector. In this case, the connector itself encapsulates the mediating behaviour of C.

We considered both cases, because our main goal is the abstraction of the relationships between the classes in the connectors. But in some cases, it is possible to find a class that can be mapped as a connector (second case).

4. Architectural Description Language as an alternative formal approach

Informal description of architecture can be efficient enough to communicate design decisions, but they have limitations to represent the real semantics of different parts of a framework. [AAG93] explains that the imprecision produced by box-line drawings makes it difficult to attach unambiguous meanings to the descriptions. It may be difficult to know when an implementation agrees with the more abstract description. It is virtually impossible to reason formally about the descriptions. It is difficult to compare two different descriptions even for the same interpretation.

Thinking in terms of giving meaning to the descriptions of software systems, Architectural Description Languages (ADLs) have been proposed to support architecture-based development, formal modelling notations and analysis and development tools that operate on architectural specifications [MT97]. An ADL must be able to communicate the architectural structures involved within a system to all stakeholders. The level of granularity, or abstraction, must be flexible enough to allow descriptions in sufficient detail or abstraction dependent on the users of the architectural description [All97]. The benefits of an architectural analysis are enhanced by precise semantics. Elimination of ambiguity is paramount in any architectural description to accurately describe a system. This requirement must be balanced with the competing goals of allowing informal descriptions [Bro00].

Wright is a formal language for describing software architecture. As most ADLs, Wright describes the architecture of a system as a graph of components and connectors. Components represent the main centres of computation, while connectors represent the interactions between components. Moreover, unlike many other ADLs, Wright also supports the explicit specification of new architectural connector types [All97]. Each part of a Wright description –port, role, computation and glue- is defined using a variant of CSP. Each such specification defines a pattern of events (called a process) using operators for sequencing ("→" and ";"), choice ("Π" and "[]"), and parallel composition ("||").

To guarantee that an architectural description is both consistent and complete, Wright provides a set of tests. We just mention them: Port/Computation Consistency (component), Connector Deadlock-Free (connector), Roles Deadlock-Free (role), Single Initiator (connector), Initiator Commits (any process), Parameter Substitution (instance), Range Check (instance), Port-Role Compatibility (attachment), Style Constraints (configuration), Style Consistency (style), Attachment Completeness (configuration). More detailed information can be obtained in [All97]. In our approach we use this set of tests to check the validity of the description we get from our algorithm, exploiting the formal side of Wright.

5. Mapping and assumptions

One of our goals is the definition of rules to define a mapping from informal documentation of frameworks to an architectural description. Thus, firstly, the path from documentation and/or code to an architectural description of a framework is reduced to a recipe. Secondly, the mapping allows us to identify micro-architectures, architectural components and connectors related to the framework.

We present, here, a defined mapping between code, written in Java or in Smalltalk, in terms of CSP process. This lets us define easily the protocol for the ports and computation of the components and the role and glue of the connectors

5.1. *Mappings : Definition and Assumptions*

In Wright, a component is defined by an interface and a computation. The interface consists of a number of ports. Each port represents an interaction in which the component may participate. A connector consists of a set of connector roles and the connector glue. Each role specifies the behaviour of participant in the interaction and the glue describes how the participants work together to create an interaction. Thus, the general structure of a component and a connector in Wright is the following one:

Component *ComponentName*
    Port $NamePort_1$ = ...
    ...
    Port $NamePort_n$ = ...
    Computation = ...
Connector *ConnectorName*
    Role $NameRole_1$ = ...
    ...
    Role $NameRole_n$ = ...
    Glue = ...

As our first objective of working in different levels of description is to be closer to the code going from one coarse-grained level to fine-grained level of description, firstly we show the mapping from the code to the architectural elements. Our mapping must be in terms of CSP process to be able to used in Wright. It must be clear enough to the reader that in most of the cases, we are constrained by the possibilities to represent architectural elements with Wright ADL.

5.2 *Mappings for Classes*

A direct mapping between classes and components is made. This means that each class in the class model is considered as a component in the description. But we decouple all the information about class communication between the component and the connector. All the information about the communication to other objects is put in the connector. For example, supposing that we have a class model to represent a book that is composed pages (See Figure 2).

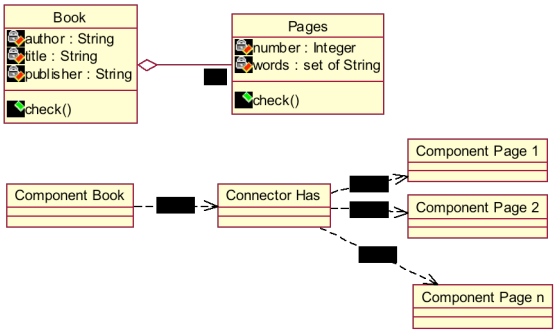

Figure 2: Class Model for a Book and the Mapping with Connectors and Components

These classes are mapped as two components: *Book* and *Page*, and we get a connector *Has* which represents the relationship between the two components. The behaviour of the component regarding to the other objects is left to the connector. In this case, for example, when the Book has to perform a spelling checking, it only sends the event to the connector which forwards the events to the pages. This is also illustrated graphically in Figure 2.

5.3 Mapping for Relationships between Classes

Besides the relationships that can be mapped by the message sendings (explained in the next subsection), we also consider three kinds of relationships between two classes A and B.

- instances of B can be instances/class variables of A
- instances of B used as parameters in one method of A
- instances of B are connected to an instance of A by a dependency mechanism.

In all the cases, we represent A, the instances/class variables of A and the parameters as components in the Wright description, whenever the objects are not instances of primitive classes (in Smalltalk) or primitive types (in Java). We only want to keep objects with a composite structure. For example, in the figure 3 we can see the class model of a Truck and its representation in terms of components and connectors. It must be clear for the user that this is a complete representation for this model, this means that we can simply represent the Truck and avoid any information of the Manufacturer. But it is clear that we do not have, for example, a component for the name of the Manufacturer. All the information (called as simple) can be used as parameters. This assumption is taken because the management of parameters in Wright is limited to simple parameters such as letters and integers.

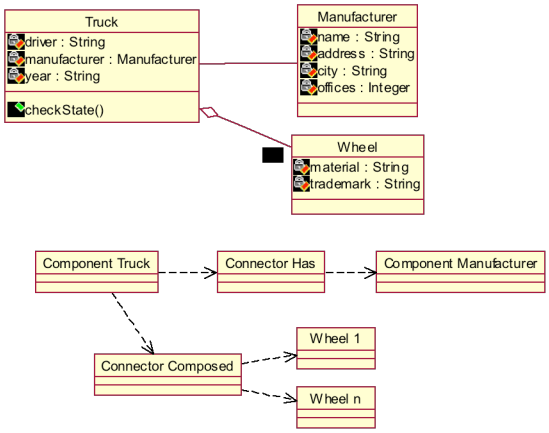

Figure 3: Class Model for a Truck and Representation with Components and Connectors

5.4 Format for Components and Connectors

As our approach focuses on having components only as units of computation and connectors as units of communication, we model the classes as components without any knowledge of what objects are connected with and we leave all this information to the connectors. In this way, the names of the ports in the component are left to the user, but we adopt the names of the roles in the connector with the names of classes that must be connected. If an instance of class A must be connected to instances of class B, the connector have the following format:

Connector AB
Role A =....
Role $B_{1..n}$ = ...
Glue = ...

5.5 Mapping for the Messages

The method calls in a method m_k in a class A have the following formats:

- in Smalltalk:
  - $object_i\ message_j$
  - $object_i\ message_j: p_1\ with: p_2 \ldots with: p_n$
- in Java:
  - $object_i\ .\ message_j$
  - $object_i\ .\ message_j\ (p_1, p_2, \ldots p_n)$

where $object_i, p_1, p_2, \ldots p_n$ can be instances/class variables of A, parameters in the method or the class itself and $message_j$ can be seen as a service that the class initializes or simply the notification of a change (dependency mechanism). We are assuming that objects p_i have only one level of objects' composition in their

structure. Following with the example of the Truck (Figure 3), p_1 can be an instance of the Wheel class, but it cannot be an instance of the Truck Class, because this latter has two levels of composition. We are making this assumption because we are interested in being able to decompose the parameter p_i in terms of its components $p_x(o_{x1},..,o_{xm})$. In the case of Manufacturer, we will get *manufacturer(name, address, city, offices)*. Thus, we get a good level of expressiveness in the description. Based on the structure of methods, let us see the different focus that we have about the methods themselves and their bodies. Firstly, all the methods (m_k) are mapped as events in CSP. Thus, we must know if the methods are called by the another object or if the method is a 'shooter' of actions. In the first case, the method will be mapped as an **observing event** m_k, and in the second case, the method will be mapped as an **initiating event** $\overline{m_k}$. So the component A (class A mapped as component in the description) has the following structure:

Component *A*

Port $Out = (m_k \mid \overline{m_k}) \mid$

$(m_k \mid \overline{m_k} \rightarrow \overline{message_j ! objectName} \rightarrow \forall x \in 1..n; \overline{asParameter ! p_x(o_{x1},..,o_{xm})}$

Computation $= (Out.m_k \mid \overline{Out.m_k}) \mid$

$(Out.m_k \mid \overline{Out.m_k} \rightarrow \overline{Out.message_j ! objectName}$

$\rightarrow \forall x \in 1..n; \overline{Out.asParameter ! p_x(o_{x1},..,o_{xm})}$

Connector *AB*

Role $A = (m_k \mid \overline{m_k}) \mid$

$(m_k \mid \overline{m_k} \rightarrow \overline{message_j ! objectName}$

$\rightarrow \forall x \in 1..n; \overline{asParameter ! p_x(o_{x1},..,o_{xm})}$

Role $objectName = message_j \rightarrow \forall x \in 1..n; asParameter ? p_x(o_{x1},..,o_{xm})$

Glue $= (\overline{A.m_k} \mid A.m_k) \rightarrow A.message_j ? objectName$

$\rightarrow \forall x \in 1..n; A.asParameter ? p_x(o_{x1},..,o_{xm}) \rightarrow \overline{objectName.message_j}$

$\rightarrow \forall x \in 1..n; \overline{A.asParameter ! p_x(o_{x1},..,o_{xm})}$

The connector uses a name matching with the parameter *objectName* and thus identify to which component (identified with the portname) it sends the events.

5.6 Mapping for Classes Creation

Class A must create instances of classes B in one of its methods, so we find the following:

- in Smalltalk:
  - *B new*
  - *B new: p_1 with: p_2 ... with: p_n*
- in Java:
  - *B ()*

- $B\ (p_1, p_2, .., p_n)$

In the first case, it maps as a special event name $\overline{create!B}$ in the protocol of the component. The class A is creating an element B so it is an initiating event. In the second case, it maps as a sequence of events

$$\overline{create!B} \rightarrow \forall x \in 1..n; \overline{asParameter!\, p_x(o_{x1},..,o_{xm})}$$

5.7 Mapping for Conditional Statements

In the method calls we can also have conditional statements.

- in Smalltalk:
  - *(condition) ifTrue: [actionTrue]*
  - *(condition) ifFalse: [actionFalse]*
  - *(condition) ifTrue: [actionTrue] ifFalse: [actionFalse]*
  - *(condition) ifFalse: [actionFalse] ifTrue: [actionTrue]*
  - *(condition) whileTrue: [action]*
  - *(condition) whileFalse: [action]*
  - *1 to: n do: [action]*
- in Java:
  - *if (condition) actionTrue*
  - *if (condition) actionTrue else actionFalse*
  - *while (condition) action*
  - *for (i:=0; i++; i<=n) action*

where the condition can only be one boolean expression (e_1) or a set of boolean expressions ($e_1,..,e_n$) joined by logical operators (*and, or, xor*), and the *action*, *actionTrue* and *actionFalse* can be a method call (m_1) or a sequence of method calls ($m_1,..m_k$). In the case of expressions, they are method calls which reply True or False. Thus in both cases, we consider them as events inside the description.

We start with the condition. The expressions $e_1,..,e_n$ are a sequence of method calls (except in the case they evaluate an internal state of the object, e.g. comparing two values of instances variables), so we map the condition such as:

$$Process\ \textbf{Condition} = \forall x \in 1..n; \overline{e_x} \rightarrow (answer?True \rightarrow ...\ []\ answer?False \rightarrow ...\ )$$

We leave the responsibility of evaluating the logical expression to the connector. Let us see how the process in the connector would be modelled.

Connector *Logic (nb: 1..n)*

Port *A* = ***Condition***

Port $B_{1..nb}$ = $e_j \rightarrow (\ \overline{answer!True} \rightarrow B\ []\ \overline{answer!False} \rightarrow B\ )$

Computation = $\forall x \in 1..n; (A.e_x \rightarrow \overline{B_x.e}\ )$

if the logical operator is 'and' :

$$\forall x \in 1..n; B_x.answer\,?\,True \rightarrow \overline{A.answer!True}$$

$$[]\ \exists x \in 1..n; B_x.answer\,?\,False \rightarrow \overline{A.answer!False}$$

if the logical operator is 'or' :

$$\exists x \in 1..n; B_x.answer\,?\,True \rightarrow \overline{A.answer!True}$$

$$[]\ \forall x \in 1..n; B_x.answer?False \rightarrow \overline{A.answer!False}$$

We consider the possibility of expressing the condition explicitly. But if the user decides to avoid it, the events to execute when the condition is true or false are expressed as non deterministic choices, because we do not have any information about the condition, and we can think that the component takes the decision of the actions to follow. In the other case, we must use the deterministic choice, because it has an observing event (*answer*) which communicates the result of the condition.

The if-statement maps as a non deterministic choice with the following format:

$$Process\ \textbf{\textit{if}} = \overline{actionTrue}\ (or\ \forall x \in 1..n; \overline{actionTrue_x})$$
$$\Pi\ \overline{actionFalse}\ (or\ \forall x \in 1..n; \overline{actionFalse_x}$$

But in the case of using the condition explicitly:

$$Process\ \textbf{\textit{if}} = all\ the\ conditions\ are\ sent \rightarrow$$
$$(answer?True \rightarrow \overline{actionTrue}\ (or\ \forall x \in 1..n; \overline{actionTrue_x})$$
$$\square\ answer?False \rightarrow \overline{actionFalse}\ (or\ \forall x \in 1..n; \overline{actionFalse_x})$$

In the case of a WHILE statement, using non-deterministic choice:

$$Process\ \textbf{While} = \overline{action}\ (or\ \forall x \in 1..n; \overline{action_x}) \rightarrow \textbf{While}\ \Pi\ \S$$

The For statement maps as a non deterministic choice with the following format:

$$Process\ \textbf{\textit{For}} = \overline{action}\ (or\ \forall x \in 1..n; \overline{action_x}) \rightarrow \textbf{\textit{For}}\ \Pi\ \S$$

The deterministic approach for While and For are similar to the one presented with the statement IF.

6. Methodological implementation

Once the mappings defined, we propose, here, some methodological steps to infer an architectural description of the framework in terms of components and connectors explicitly, based on the mappings and the documentation we can have on of the framework (class hierarchies provided in UML, source code). One objective is to work at different levels going from domain-specific to implementation-specific issues.

Step 1: Identify the main classes of framework in terms of the domain. This step is concerned with identifying classes which were mapped to concepts of the studied domain. In most cases these classes are clearly identified in the design. If we have class hierarchies, we suggest to take the root class of the hierarchy.

Step 2: Each class is mapped to a component and each possible relationship between two classes is mapped to a connector in terms of Wright, avoiding to have a relationship with classes of simple types (integer, char, boolean). This step is concerned with getting relationship between classes which are composed of other objects.

Step 3: The protocols of each class are classified as initiating or observing events, and all the messages called in the body of the messages are classified as

initiating events. We avoid to take into account methods classified in protocols categories such as initializing or accessing, and also the assignments in the implementation.

Step 4: The protocols for the ports and the computation of the components are built.

Step 5: The connectors are built using the messages sent from one class to another one.

Step 6: Identify the variations of the one component (each subclass of a root class) and what other components related to the component must be changed. In the first step we identify abstract classes, in the case of class hierarchies, in order to have the components in the first description. But this component is just one prototype of other components that can be mapped from the subclasses. Thus, if we have a class hierarchy, the idea is just to take each subclass, to see what other classes are related and then map them as components. Then, repeat the process from the step 3 until to get different versions of the description.

Step 7: Identify the components that represent hotspots and frozen spots. This step is focused to identify which components and connectors are fixed (this situation can be detected in the different descriptions obtained from Step 6) and which ones are candidates to be changed in terms of a framework instantiation

Step 8: At this step, we have a first level of description. We can identify predefined architectural styles in terms of set of classes, or just components and connectors with a specific behaviour.

Step 9: We run the tool to check the different properties in Wright (Port Computation Consistency (component), Connector Deadlock-Free (connector), Roles Deadlock-Free (role), Single Initiator (connector), Initiator Commits (any process), Parameter Substitution (instance), Range Check (instance), Port-Role Compatibility (attachment), Style Constraints (configuration), Style Consistency (style), Attachment Completeness (configuration)).

This step only ensures us that our description is valid using Wright. If there is an error reported by the tool, we should check it following the format defined in the Steps described previously.

Step 10: Refine each component considering two cases:

- if we have a hierarchical composition of objects that work together (definition of micro-architectures), then the goal of this step is to discover if there is a component that is composed of other objects, and the different services that it offers are made using these objects. All the objects must be inside the 'boundaries' of the main object to consider it as a hierarchical composition.
- If there is a set of events joined by a non-deterministic choice which indicates a decision of the component regarding an internal state (internal state of the component), then this step is concerned with expressing all the information related to the component avoiding to have non-deterministic choices.

Step 11: New components (not necessarily mapped from domain concepts) can be discovered. This step is concerned with having a new level of description. If this situation happens, it is suggested to start to study the component as a micro-architecture and to follow again the steps only with the new components.

If we are interested in refining the behaviour of the component, this step is also concerned with having detailed information about the defined behaviour protocol in the components.

Step 12: Definition of the interaction protocols in interface types and association of frozen spots and hot spots in styles. This step is concerned with identifying the set of events that belong to an interaction protocol and defining styles for the fixed part and variable points of the framework. This will allow us to have a clear view of how the framework is composed and measures the impacts of possible changes in its structure and object behaviour.

[Are00] shows two examples of the applicability of the algorithm, the mappings and the results obtained.

7. Some Results

The use of an ADL to describe an object-oriented architecture has more advantages than drawbacks.

However, one major problem encountered is to not being able to have the internal state of a component. In fact, Wright focuses on the interaction behaviour and there is no possibility to use the operations given by the components. Most of the executions are hidden behind the non-deterministic choices in the specification. Thus, it is not possible to have any specification about the functional aspects of the components. [San97] proposes the use of Abstract Machines B to solve this problem. Another difficulty concerned the addition of a management of errors. Due to the static nature of Wright, to do this we must change the interaction protocol of the components and connectors and we lose the expressiveness of the protocols because the behaviour and the error messages are mixed. We can also mention that the hierarchical relationship between two classes is not explicit in the description. If we change a component from one level of description to another one, and the latter component represented a subclass of the class mapped as the first component, this relationship is lost.

We applied our work on an object-oriented framework [Bos00] for which we had the code and the design, the information about semantic structure was almost hidden. We got an architectural description with the following characteristics:

- The classes mapped as combinations of components and connectors. We obtain two models: units of computation (components) and units of communication (connectors).
- The model of communication (set of connectors) providing all the framework behaviour, and making explicit the method call ordering and the interaction protocol between the different classes.
- A classification of the messages inside a class: by using the concept of initiating and observing events of Wright, we were able to differentiate class dependent messages from messages called by other classes.

In [Are00] we show how we use architectural styles to express the fixed parts of a framework (hotspots), and we get a formal model that characterises all the applications resulting from the framework instantiation.

8. Conclusions

The lack of established rules to define how we can infer an architecture from an application in general and the treatment of software architectures in a 'high level of abstraction' made us to define the constraints of our study context. We restricted our work at a low-level notion of software architecture that reflects the semantic structure of a software system: the code level combined with information on the design level of the frameworks.

In this paper we have presented the definition of the mapping from Java/Smalltalk code to CSP notation and a set of steps showing one way to get an architectural description for an object-oriented framework. Real examples are treated in [Are00]. We think that our description technique can be used in conjunction with other techniques as a complement, since this is a way to provide a 'bridge' between informal and formal approaches.

It could be valuable to extend this work with an ADL which allows dynamic architecture representation. For example, to represent the evolution of an architecture and configuration changes during execution. New languages go in this direction: Piccola [ALSN98], π-space [CGOW00], dynamic Wright.

At last the development of semi-automatic tools to support the mapping and the methodology would be useful to add more validations.

9. References

[AAG93] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM SIGSOFT Software Engineering Notes*, 18(5):9-20, December 1993

[ALSN98] F.Achermann, M.Lumpe, J-G. Schneider, O.Nierstrasz, PICCOLA - a Small Composition Language, Software Composition Group, University of Berne, 1998.

[All97] R. Allen. *A Formal Approach to Software Architecture*. Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997

[Are00] G. Arevalo. Object Oriented Architectural Description of Frameworks. EMOOSE Thesis, Ecole des Mines de Nantes, Nantes, France and Vrije Universiteit Brussel, Brussels, Belgium. 2000

[Bos00] J. Bosch, Measurement Systems Framework, *in Domain specific Application Frameworks: Frameworks Experience by Industry*. M.E. Fayad, R.E.Johnson editors, Wiley 2000.

[Bro00] L. Bross. *Box Structures as an ADL*. Master Thesis to appear. University of South Florida, 2000. http://home.tampabay.rr.com/adls/archben.html

[CGOW00] C.Chaudet, M. Greenwood, F.Oquendo, B.Warboys, A Formal Language for Describing Evolving Software Systems: Architectural Compositionality and Evolvability, Second Workshop on Object-Oriented Architectural Evolution, ECOOP'2000.

[FSJ 99] M.E. Fayad, D.C.Schmidt, and R.E.Johnson, Building Application Framework, Wiley 1999.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6): 17-26, November 1995

[Hoa85] C. A. R. Hoare. *Communicating Sequential Programming*. Prentice Hall, 1991

[Mat96] M. Mattson. *Object Oriented Frameworks: a survey of methodological issues*. Licentiate Thesis, Department of Computer Science, Lund University,1996

[MB00] M. Mattsson and J. Bosch. Object oriented frameworks: Composition problems, causes and, solutions. In *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, pp. 467-487, M. Fayad, D. Schmidt, R. Johnson editors, Wiley Press, 2000

[MKMG96] R. T. Monroe, D. Kompanek, R. Melton, and D. Garlan. Stylized Architecture, Design Patterns, and Objects. *IEEE Software*, Jan 1997, pp. 43-52

[MN95] T. D. Meijler and O. Nierstrasz, Beyond Objects: Components, *Cooperative Information Systems: Current Trends and Directions*, M.P. Papazoglou and G. Schlageter (Eds.), pp. 49-78, Academic Press, November 1995

[MT97] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97),* pp. 60-76, Lecture Notes in Computer Science Nr. 1013, Springer-Verlag, September 1997

[Ric98] T. Richner. Describing framework architectures: more than Design Patterns. In Proceedings of the ECOOP '98 Workshop on Object-Oriented Software Architectures, *Lecture Notes in Computer Science Nr. 1543,* J. Bosch and S. Demeyer editors. Springer-Verlag, 1998

[San97] R. Sanlaville. *Description d'architecture logicielles: Utilisation du formalisme Wright pour l'interconnexion de machines abstraites B. Report for DEA d'Informatique: Systems et Communications*. Laboratoire LSR (Logiciels, Systèmes, Réseaux). Université Joseph Fourier. Grenoble. France. 1997

[SG99] J. P. Sousa and D. Garlan. Formal modeling of the enterprise JavaBeans component integration framework. In *Proceedings of FM '99, Lecture Notes in Computer Science 1079*, Toulose, France, September 1999. Springer-Verlag

[UML97] Rational Rose. UML Notation Guide Version 1.1. September 1997. http://www.rational.com/UML