Mapping an ADL to a Component-based Application Development Environment*

Virgínia C. C. de Paula and Thais V. Batista

Department of Informatics and Applied Mathematics - DIMAp
Federal University of Rio Grande do Norte - UFRN
{vccpaula,thais}@ufrnet.br

Abstract. In this paper we discuss the mapping of an Architecture Description Language, ZCL, to an environment for configuring distributed application using CORBA components, LuaSpace. We focus on the mapping of the structural properties of ZCL and LuaSpace. In order to deal with compatibility issues, we propose an extension to ZCL. The result of this work is an integrated environment that combines the flexibility and execution plataform provided by LuaSpace with a tool for design and for consistency checking - ZCL. The experience in combining an ADL with a configuration based environment can give clues on how integrating the activities of design and implementation during the lifetime of a software system.

keywords: Software architecture, configuration, component, CORBA, ADL, scripting language, dynamic reconfiguration.

1 Introduction

Component-based development is a current trend in software engineering mainly because it promises to concretize the idea of reusing existing components by pluging-and-playing them in order to compose an application. Frameworks for component interoperability are playing an important role in component based application development because they offer support for working with heterogeneous components despite differences in language and in the execution platform. The CORBA model [22] has drawn special attention as a framework for interoperability because it is independent of language and manufacturer and provides easy access to components with transparent distribution. However, CORBA, like other frameworks for component interoperability, does not have facilities to describe the global organization of an application [25].

Several development environments [25, 6, 24] have been proposed to support the construction of component-based applications. In general these environments are based in a two level development model, in which not only the *component*

* Research for this paper was done with valuable support from CNPq (Brazilian Council for Development of Science and Technology) under processes 68.0103/01-5 and 68.0102/01-9.

level is considered but also the *configuration level*. In this second level, the structure of the application is composed of components and their interaction. Both level support the implementation phase of a software application.

Another important field that has gain an momentum in component-based development is *Software Architecture* [14]. The concept of software architecture, also said system structure or system configuration, is especially important to design complex software systems, providing a model of the large scale structural properties of systems. These properties include the decomposition and interaction among parts as well as global system issues such as coordination, synchronization and performance [15]. Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication [14]. Therefore, it adresses the high-level design of a system [21], named *Architecture level*. In this level the correct characterization of component composition and relationship among components are defined using a Software Architecture Description Language (ADL).

Although the terms *architecture* and *configuration* are usually treated as synonymous in software architecture literature, in this work we call *architecture* the abstract level via which we model an application. The term *configuration* we use when talking about a level close to the application implementation.

In [21], it is mentioned that software architecture and configuration based programming are related research areas that evolved separately but the solutions in these areas center around the same system model. Despite some differences, the approaches are complementary and share some features such as a related terminology and the notion of components and configuration.

As mentioned in [16], some ADLs *are taking a more pragmatic approach to the development of distributed systems*. This approach is worried about dealing with middleware infrastructures, such as CORBA.

In this paper, we describe the mapping of an ADL, named ZCL [1, 3] to a component-based application environment - LuaSpace [24]. The motivation that lead us to develop this work arise from both sides involved. From the perspective of ZCL, it is necessary an underlying execution platform to run the application described using ZCL and to avoid *architectural erosion* [21]. From the perspective of LuaSpace, the use of an ADL can document the application design phase, give an overall vision of the application topology and guide the application consistency maintance. In a more generic way, we want to integrate the flexibility and execution platform provided by LuaSpace with a tool for design and for consistency checking - ZCL.

One of the main features of ZCL is to have an associated formal framework [1], specified in Z [4], to describe and reason about dynamic distributed software architectures. It focuses on the operations necessary for the construction of dynamic software architectures. ZCL deals with execution issues defining states for the components and connections. So, the architect can concentrate on architectural issues or he/she can also analyse execution issues. Although ZCL considers execution issues, a platform is not available do run applications specified using ZCL. Nevertheless, simulations are allowed using Z tools.

The main advantage of LuaSpace is to provide a flexible way to compose application using CORBA components and to support dynamic reconfiguration of applications. LuaSpace uses an interpreted and procedural language - Lua - and a set of tools based on this language as a configuration tool to programming a component-based application. LuaSpace focuses on flexibily for dynamic reconfiguration.

Architecture descriptions and middleware are used in different phases of software development. The decision of modelling an application to be executed using an specific middleware is a typical case in which the implementation has direct influence on architecture decisions. Therefore, we intend to provide an appropriate way to model an architecture to be executed in LuaSpace, because *for a system to be implemented in a straightforward manner on top of a middleware, the corresponding architecture has to be compliant with the architectural constraints imposed by the middleware* [16].

ZCL and LuaSpace were originally thought in a completely independent way. However, they share a common interest in dynamic reconfiguration, supporting dynamism in different abstration levels. The integration must keep unharmed the advantages of both approaches and it may enhance them with new functionality.

This paper is structured as follows. Section 2 presents the background of this work describing LuaSpace and ZCL. Section 3 discusses their integration. Section 4 presents related works. Finally, section 5 presents our final remarks.

2 Background

2.1 ZCL

The ZCL language is based on the CL language [11, 12], which uses most of the principles of other MILs, but it has introduced new concepts, like planned reconfiguration. In this kind of reconfiguration, the designer can predict some modifications as likely to happen.

In ZCL, an architecture has a hierarchical structure in which the architecture is a composition of components that can also be composite. Those components that implement a functionality are simply called *components* or *task components*. *Composite components* can be seen as (sub)architectures and are also called *group components*. *Task components* are the smaller unit of computation we are considering. An architecture in ZCL is constructed by successive use of its operations. The components must exist in the library of components to be used in the description of an architecture. ZCL includes (auxiliary) operations to add components to the library.

Task components, composite components and ports of communication are the basic elements of an architecture in ZCL. Communication ports constitute the interface of a component through which it communicates to other components. A *link* is a connection between two communication ports.

One important feature of ZCL is to have an associated formal framework [1, 3], by which the software architect can describe and reason about dynamic

distributed software architectures. ZCL was specified using Z[4]. In order to illustrate a Z schema, we present below the *ZCL_Component* schema which represents a task component. It specifies the *interfaces* as a set of *PortNames*. Every port has *Port_Attributes*: direction (*DIR*) to indicate that it receives messages (entryport) or sends messages (exitport); mode (*MODE*) to indicate that it can be *notify* (asynchronous) or *requestreply* (synchronous); and type (*TYPE*) to indicate the type of data that can be transmitted by the port. It is also possible specify application specific attributes of a component (*component_attr*). The given sets *Indices* and *Attributes* are used to classify application specific attributes. *ID_Component*, *Nodes*, *PortNames* and *Location* are given sets representing respectively identifiers of components, of instances, of ports and of machines in which instances are executed. Each element in ZCL is specified by a schema in Z. So, we have schemas specifying components, composite components, instances of components (or composite components), etc.

$$
\begin{array}{|l}
\hline
\textit{ZCL\_Component} \\
\textit{component\_attr} : \textit{Indices} \rightarrow\!\!\!\!\!+ \textit{Attributes} \\
\textit{interfaces} : \mathbb{F}_1\, \textit{PortNames} \\
\textit{port\_attr} : \textit{PortNames} \rightarrow\!\!\!\!\!+ \textit{Port\_Attributes} \\
\hline
\mathrm{dom}\, \textit{port\_attr} \subseteq \textit{interfaces} \\
\hline
\end{array}
$$

ZCL works with the concept of *library of components*. Components existing in the library can be *used* in the context of an architecture description. Instances of components can be *created* and *linked* to other instances. They must have the same interface to be linked. In ZCL, instances are not automatically activated. This allows better control over parallelism. The architect must explicitly say that an instance has to be activated. In summary, we can say that ZCL allows the following commands to create an initial architecture: *use*, *create*, *link* and *activate*. The commands to allow reconfiguration are: *remove*, to remove a component from the context; *delete*, to delete an instance; *unlink*, to disconnect ports; and *deactivate*, to stop an instance. Moreover, each link is associated to a *ZCL_Connection*, which stores information about each pair of ports connected. In ZCL, an instance of port is always associated with an instance of component (*Nodes*). Each connection has a buffer in which messages exchanged between the sender (output port) and the receiver (input port) are stored.

Static verifications can be carried automatically by the framework. For example, to verify whether a component being declared by the architecture exists in the library or to verify whether an activate command is referring to an instance already created.

As said above, ZCL focuses on the operations necessary for the construction of dynamic software architectures. Each operation to mount an architecture has a corresponding one to annul its efect. Therefore, an architect can modify an architecture, but ZCL assures that the modification is done just in case it is a valid one (it leaves the architecture in a valid state). To do that, the ZCL framework has also an execution model based on states, which has the responsibility

for verifying whether an architecture is in a valid state. So, the architect can concentrate on architectural issues or he/she can also analyse execution issues.

The architecture being described is stored in the *configuration table*, which is dynamically modified to reflect changes suffered by the application architecture. The ZCL operations use this table to ensure that the application is in a state suitable for modifications. All operations contain error cases also specified as schemas in Z. When any constraint of the operation is not obeyed, the error case schema of the operation is used.

As said in [13], if an architectural fact is not explicit in the architecture, or deducible from the architecture, then the fact is not intended to be true of the architecture. Therefore, it is extremely important to have a model in which all relevant features of an architecture can be specified. Observe that the designer can use ZCL to both analyse static architectures and run-time issues, such as dynamic reconfiguration. Observe also that the ZCL framework is highly modular and we have separated the schemas related to structural (static) analysis from those related to dynamic analysis. This means that the execution model based on states can be easily replaced or modified.

2.2 LuaSpace

LuaSpace is an environment for development of component-based applications integrating the CORBA platform with the interpreted and procedural language Lua [19], used to glue components. The application is written in Lua and can be composed by components implemented in any language that has a binding to CORBA.

LuaSpace provides a set of tools based on Lua that offer strategic functions to facilitate the development of component based applications and to promote dynamic application configuration. These tools are: LuaOrb, Generic Connector, Meta-Interface and ALua [18]. Next, Lua, LuaOrb and Generic Connector are briefly presented. The other tools are not presented here because they are not necessary in the context of this work.

- *Lua* [19] is an dynamically typed, interpreted and procedural configuration language that integrates strong data description facilities and reflexivity with a simple sintax. Lua includes conventional aspects, such as syntax and control structures similar to those of Pascal. It also has several non-conventional features: functions are *first-class* values; associative arrays (called *tables* in Lua) are the single data structuring facility; *tag methods* are Lua's most generic mechanism for reflection. Tag methods can be specified to be called in situations in which the Lua interpreter does not know how to proceed. It is the base for the implementation of the tools that compose LuaSpace because in LuaSpace when the Lua intepreter does not know execute a command, it invokes the appropriate tag method that knows to handle the command.
- *LuaOrb* [26] is a binding between Lua and CORBA based on CORBA's Dynamic Invocation Interface (DII) that provides dynamic access to CORBA components available at remote servers exactly like any other Lua object.

This is done transparently and at runtime. Moreover, it uses CORBA Dynamic Skeleton Interface (DSI) to permit dynamic installation of new objects in a running server.

- the *generic connector* [23] is a mechanism to configure an application as a set of services without being aware of the specific components that implements the service. Those components, if available, would be inserted into the application during its execution. At runtime, the generic connector searches for components that can provide the service stated in the application configuration program, activates the service and returns the result to the client. This mechanism introduces a great flexibility in application modelling since the developer can abstract away about specific components. It also addresses dynamic reconfiguration because different invocations of the same service may result in the selection of different components. With the use of the generic connector, it becomes imposible to distinguish the tasks of configuration and of reconfiguration in the configuration program.

To illustrate development using LuaSpace, we present the producer-consumer application. This application consists of *producer* and *consumer* components, whose IDL interfaces are illustrated in Figure 1. The *producer* component periodically adds an item in a buffer and can removes items from buffer. The *consumer* component can retrieve an item from the same buffer when it receives a notification that there is an item to be consumed. It can also ignore the notification. Issues regarding the concurrency control of the producer and consumer application are not in the scope of this work. Figure 2 shows an example of a configuration program developed using LuaSpace to the producer-consumer application. There are three *consumers* that receive a notification (`receive_note` method) when an item is produced (`produce` method) and eventually it is interested in retrieve an item (`retrieve(item = ``OK'')`) from buffer. In this case, `remove` method of the producer is invoked.

```
interface producer{
   string produce(item);
   void remove(item);
}

interface consumer{
   void receive_note();
   string retrieve(item);
}
```

Fig. 1. Producer-Consumer IDL Interfaces.

The use of an interpreted and procedural configuration language introduces a different style of configuring an application where explicit linking and unlinking commands are not necessary. The procedural model is used to describe an

```
i = 1
p = createproxy{``producer''}
while i<=3 do
    c[i] = createproxy{``consumer''}
    i = i + 1
end
i = 1
while true do
   if (p:produce(item) = ``OK'') then
      while i<=3 do
          c[i].receive_note()
          if (c[i].retrieve(item) = ``OK'') then
              p:remove(item)
          end
          i = i + 1
      end
   end
end
```

Fig. 2. LuaSpace configuration program.

application. As a consequence there is no rigid distinction between the configuration and reconfiguration of an application. The simple use of conditional (if) or iteration (while) command in the configuration program implies in dynamic reconfiguration. Through an interactive console is possible to directly build and modify configuration programs. As Lua is a dynamically typed language, it is not necessary previously to declare the component instances that will be used in a program. New components can be selected dynamically according to runtime conditions.

LuaSpace provides support to implement both programmed and ad-hoc reconfiguration. For programmed reconfiguration, the Lua conditional commands can be used to establish the conditions that determine reconfiguration in the application source code. In this way, reconfiguration points are explicitly defined by the programmer. Moreover, the generic connector introduces the possibility of automatic reconfiguration, since for each call, different components can be selected to execute the required service.

Ad-hoc reconfiguration can be programmed interactively through the Lua console. This tool offers the programmer a way to have control over the application by directly interacting with the system. Another way is to use the ALua mechanism in which reconfiguration can be defined by sending the application a message with the reconfiguration code to be executed.

LuaSpace can be used in two scenarios of applications. In one scenario, LuaSpace is used to develop applications that explores the flexibility in lieu of static checking and that are not worried with consistency. This is the original proposal of LuaSpace because these features fit in Lua dynamic style that promotes flexibility. In the other scenario, LuaSpace is used in applications that

need to mantain integrity. In this case, a software architecture should guide the evolution of the application. Dynamic reconfiguration is done according the architectural model, following the restrictions imposed. The use of LuaSpace in this kind of application is useful because the application can explore the set of tools to facilitate dynamic reconfiguration, access to CORBA components and object localization regardless programmer intervention and, at same time, it has the guarantee that the architectural model is preserved. This work address the second senario, integrating an ADL to the LuaSpace environment. In this context, a specific ADL, ZCL, will support the semantics of reconfiguration in order to verify the validity of the changes.

3 Mapping ZCL to LuaSpace

Figure 3 illustrates the two stages of development involved in this work: ZCL at design phase and LuaSpace at implementation phase.

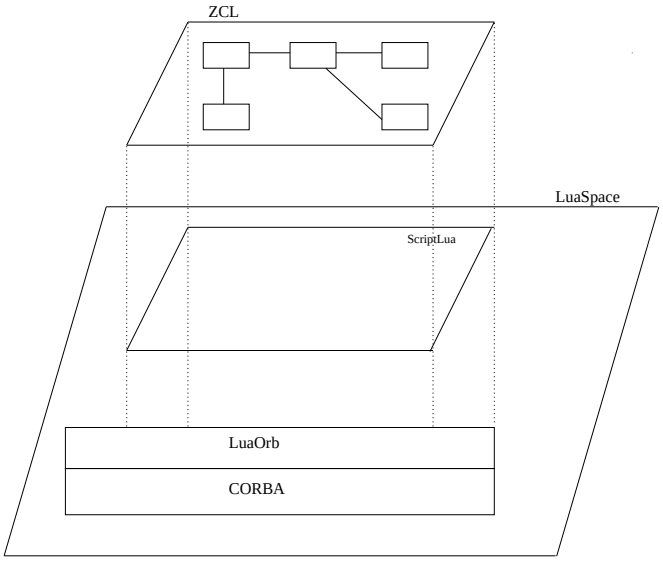

Fig. 3. Overall view of development phases

The mapping from ZCL to LuaSpace consists of defining the conversion of ZCL architectural elements to LuaSpace configuration aspects. There are significant structural differences between ZCL and LuaSpace. While ZCL follows the structure proposed by the configuration paradigm [10], LuaSpace does not obey any specific structural model. With this mapping it is possible to move from the high level specification toward the corresponding implementation.

In LuaSpace, a configuration program of an application consists of components, scripts and glue codes. Components are CORBA entities whose interface is described using the CORBA Interface Description Language (IDL). No information about implementation issues is published in component interface. Interface describes only component provided services. An application is composed of CORBA components that communicate through the CORBA bus that acts as an intermediary between components.

Since LuaSpace uses a procedural and interpreted language as a configuration tool, there is no explicit linking and unlinking command. Interconnection between components are represented by method invocation. In the same way, there is no explicit reconfiguration command. Conditional and interaction structures can determine dynamic reconfiguration. Another way of dynamically reconfiguring an application is interactively via the Lua console. In this way, there is no rigid distinction between configuration and reconfiguration in a program.

$$
\begin{array}{l}
\textit{ZCL\_ExplicitConnector} \\
\hline
\Xi\, \textit{ZCL\_Library} \\
\textit{ZCL\_Component} \\
\textit{sender} : \textit{ID\_Component} \rightarrow \mathbb{F}_1\, \textit{InteractionPoints} \\
\textit{receiver} : \textit{ID\_Component} \rightarrow \mathbb{F}_1\, \textit{InteractionPoints} \\
\textit{type} : \textit{TYPE} \\
\textit{buffer} : \mathrm{seq}\, \textit{MSG} \\
\textit{behaviour} : \textit{BEHAVIOUR} \\
\hline
\mathrm{dom}\, \textit{sender} \subseteq \mathrm{ran}\, \textit{tasks} \lor \mathrm{dom}\, \textit{sender} \subseteq \mathrm{ran}\, \textit{groups} \\
\mathrm{dom}\, \textit{receiver} \subseteq \mathrm{ran}\, \textit{tasks} \lor \mathrm{dom}\, \textit{receiver} \subseteq \mathrm{ran}\, \textit{groups} \\
\forall\, \textit{idcomp} : \textit{ID\_Component} \mid \textit{idcomp} \in \mathrm{dom}\, \textit{sender} \\
\bullet\ \textit{sender}(\textit{idcomp}) \subseteq (\textit{tasks}(\textit{idcomp}))\mathrm{.}\textit{interfaces} \lor \\
\qquad \textit{sender}(\textit{idcomp}) \subseteq (\textit{groups}(\textit{idcomp}))\mathrm{.}\textit{interfaces} \\
\forall\, \textit{idcomp} : \textit{ID\_Component} \mid \textit{idcomp} \in \mathrm{dom}\, \textit{receiver} \\
\bullet\ \textit{receiver}(\textit{idcomp}) \subseteq (\textit{tasks}(\textit{idcomp}))\mathrm{.}\textit{interfaces} \lor \\
\qquad \textit{receiver}(\textit{idcomp}) \subseteq (\textit{groups}(\textit{idcomp}))\mathrm{.}\textit{interfaces} \\
\hline
\end{array}
$$

In ZCL, components, connections and architectures (composite components) are considered architectural elements. Components can be viewed as a black boxes entitieswith a well defined interface described by input and output ports. Components interact via ports and message passing is the only way of communication between components. Therefore, in the original version of ZCL there is not explicit connector. Nevertheless, it exists connections between components ports. An architecture in ZCL is a composite component which joins all the components or other composite components and the connections between them. Modelling LuaSpace's glue codes as ZCL connections does not seem a correct decision, because, as we have already said, it limits the communication between components to message passing. Another important issue is the lack of declaration of required services in CORBA component interfaces. Therefore, we propose an extension to ZCL in which a new architectural element is created - an explicit

connector. As a consequence, we propose ports be replaced by *interaction points* at ZCL_Component schema and Port_Attributes to be replaced by *Properties*. These changes give flexibility to the framework because it is now allowed other forms of communication besides message passing.

The ZCL_Library is a schema specified by a function, *tasks*, which maps an identifier of a component, *ID_Component*, into a ZCL_Component and another one, *groups*, that maps an *ID_Component* into a ZCL_CompositeComponent. The same *ID_Component* can not exist in both sets.

A ZCL_ExplicitConnector is a ZCL_Component and it uses the ZCL_Library, which is not changed by the connector. So, it has a set of interaction points and it is composed of two sets of interaction points (*sender and receiver*), a *buffer* which would be used depending on the *type* of the connector, and a *behaviour* by which the connector can be described. The *type* of the connector is used to allow ZCL to deal with different kinds of connectors, such as glue codes, multicast channels, ORB [16], etc.

The framework has a schema to represent an instance of component. In the same way, we have included a schema to represent an instance of an explicit connector. We have also changed some of the operations schemas to adapt them to this new element. Special attention we dedicated to the link operation since we had a component-component connections and now we have component-connector and connector-component attachments.

In section 2.2, we present the producer-consumer application. We now model the same application using ZCL and the proposed extensions.

As we have already said, to create an architecture in ZCL, it is necessary to use the operations of the framework. Initially, we use the operations to create the *producer* and the *consumer* component, their interfaces and to update the library including them. We show below the corresponding operations invocations.

$$CL\_Create\_Task[comp\_attr? := \emptyset, itrct\_points := produce, remove]$$
$$\wedge\ CL\_Create\_InteractionPoint[itrct\_points? := produce, remove]$$
$$\wedge\ CL\_Update\_LibSimple[nc? := Producer]$$

The *consumer* is created using the same sequence of operations. Its interaction points are `retrieve` and `receive_note`.

Having included the components in the library, the architect has to use them in the context and to create their instances. In our case study, we want to create one instance of producer and three instances of consumer. For spaces reasons, we just present the operations which create the instances. In a similar way, we activate the instances.We have the following operations invocations:

$$CL\_Create\_Instance[node? := prod, component? := Producer]$$
$$CL\_Create\_Instance[node? := cons1, component? := Consumer]$$
$$CL\_Create\_Instance[node? := cons2, component? := Consumer]$$
$$CL\_Create\_Instance[node? := cons3, component? := Consumer]$$

Before linking the interaction points, we have to include the connector in the library and to create an instance of it. This is done invoking the appropriate operations as we have done to create components.

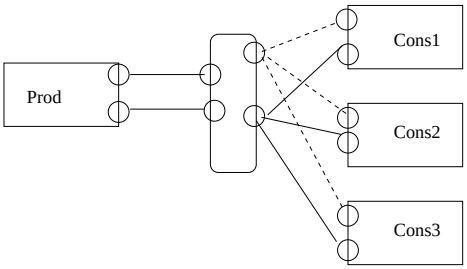

Fig. 4. Producer-Consumer

To link a component to a connector the *ZCL_Link* operation is used. It receives the pair (component instance, interaction point) and the pair (connector instance, interaction point). An interaction point is independent from the others. Therefore, the operation must be invoked as many times as the quantity of interaction points.

In Figure 4, it is illustrated the structure of the created architecture. In this Figure, we can see a dotted line which represents the invocation of the `retrieve` function by the consumer. As already said in section 2.2, sometimes the consumer is not interested in retrieving an item. In this case, we have to unlink the corresponding interaction points in the architecture. This is done by invoking the *ZCL_Unlink* operation which have the same parameters than *ZCL_Link*.

In summary, ZCL components can be considered components (or scripts) in LuaSpace. ZCL architectures model LuaSpace configuration programs that can contain components, scripts and glue codes. Component provided services of LuaSpace are equivalent to interaction points in ZCL. In LuaSpace there are many facilities to model interactions between components [20], such as: glue code, generic connector and events. In ZCL, all these elements are represented by the ZCL_ExplicitConnector abstraction, which deals with different *types* of connector, allowing the specification of the *behaviour* of them.

Dynamic reconfiguration is expressed in ZCL through operations over architectural elements, as described in section 2.1. The equivalence of ZCL operations for dynamic reconfiguration and LuaSpace reconfiguration support are execution issues and it is under development. Observe that as well as ZCL, LuaSpace addresses ad-hoc and planned reconfiguration [2].

Table 1 summarizes the correspondence between structural issues of ZCL and of LuaSpace.

4 Related Works

Although nowadays the importance of joining ADLs and Component-based development is broadly treated in several works, in this section we just present two approaches which address this issue.

Structural Issues	ZCL	LuaSpace
System Topology	Components, connectors and composite components	Components, scripts and glue codes
Interfaces	interaction points	provided services
Component interactions	Connectors	Glue code, Generic Connector and Lua event
Dynamic Reconfiguration	ad-hoc and planned	ad-hoc and planned

Table 1. Summary of Equivalences

4.1 C2

In [16], we can find a work related to our, in which the authors use ADLs to describe *middleware-induced architectural styles* and provide an evaluation of ADLs in order to show their suitability for defining middleware-induced architectural styles. The authors mention the idea of implementing, in long term, an environment that supports the definiton of architectures by providing a library of styles induced by specific middlewares. This environment would be able to partially automate the implementation of the architecture on the corresponding middleware.

The authors want to capture the architectural assumptions induced by middlewares in terms of middleware-induced styles. The middlewares Regis and C2, which ADLs (Darwin and C2SADEL) have been specifically defined, are considered. The characteristics that components, connectors and configurations of instances must have to be compliant with a specific middleware are considered to define and to specify a middleware-induced style. A similar approach considering layered style and CORBA middleware can be found in [17]. Two middlewares were chosen: JEDI and C2. They were specified using the ADLs ARMANI, Rapide, Darwin, Wright, and Aesop. In this way, some requirements ADLs should have to specify middleware-induced styles were identified [16]:

- ADLs should be able to define styles and provide a mechanism for exploiting a style in the definition of an architecture;
- ADLs must support the specification of some general topological constraints that must be respected by any specific instantiation of the component and connector types defined in the style;
- ADLs must support the description of the behaviour of components and connectors, because topological constraints are not enough to define styles;
- An important requirement for both connectors and components is the possibility of refining their internal structure in terms of the composition of other components and connectors. So, ADLs must support the co-existence of different levels of abstraction in an architecture;
- An special conclusion was taken related to connectors. Connectors have been specified explicitly. Nevertheless, in general, the semantics associated to explicit connectors does not seem appropriate to model more modern kinds of

connectors, such as event dispatchers, ORBs, and multicast channels. Therefore, it is necessary to define an intermediate, artificial connector type to attach the "real" connetors to the actual components of a C2 architecture.

Our goal is quite different from the one of the work described above in the sense we want to allow the modelling of a system to which it is known to be executed in a specific middleware - LuaSpace. More than that, we want this modelling to be specified using ZCL. Nevertheless, we can consider the conclusions listed above about ADLs to evaluate how ZCL is supposed to support "LuaSpace-oriented" specifications. We want to guarantee that the system modelled in ZCL can be implemented in LuaSpace. However, in this paper, we are not proposing an automatic implementation of the modelling system.

4.2 Darwin

Darwin [7] supports the definition of architectures in terms of components, services and bindings. Components are described by interfaces that declares the provided and required services. Systems are specified in Darwin by describing the set of component instances and the set of bindings between required and provided services.

[9] describes the use of Darwin to structure systems using CORBA objects. This work mentions that object interaction and interface compatibility are the concern of ORB and the CORBA bus and the Interface Definition Language (IDL), while the structure of an application is supported by an ADL. We think that this ADL does not address architectural mismatches [8] between components, once it supposes that interface compatibility is a issue supported by CORBA bus. CORBA bus is the mediator of the communication and facilitates transparent access to remote components but the configuration language is used to determine the components involved in the communication. We argue that there is a level - the configuration level - between the bus and the architecture description. At this level it is possible to determine the overall structure of the application and to address interface incompatibility as well as to dynamically reconfigure the application.

In the mapping from Darwin to CORBA, the Darwin compiler translates a Darwin component specification to the IDL interface. Each provision in the Darwin specification is translated into a read only attribute of the object reference type. Each requirement is similarly mapped into an attributed which is not read only because it is set externally to reflect the binding of the component instance.

In our work, there is no compiler to translate ZCL to LuaSpace. We define the mapping of each architectural entities into configuration elements.

5 Final Remarks

In this paper we evaluate the mapping of an ADL to an environment for component-based application development that uses a scripting language and

associated tools for configuring an application. We identify the common terminology of the two research areas involved: software architecture and configuration-based development. We focus on the mapping of the structural properties of ZCL and LuaSpace. We also propose an extension to ZCL in order to address its differences to LuaSpace, regarding modelling components interconnections. The definition of the behavioral aspects are under development. We are analising the correspondence of the states treated by the ZCL execution model with the LuaSpace execution issues.

Comparing the features of ZCL with those mentioned by [16] and presented in section 4.1, we can say that ZCL and its extension proposed in this paper, satisfies almost all of them. The only one ZCL still does not satisfy is the refinement of connectors. Composite components in ZCL supports refinement of internal structure of components.

The experience in combining an ADL with a configuration based environment can give clues on how integrating the activities of design and implementation during the lifetime of a software system. In [5] we find a classification of technologies as *component-centric* and *system-centric*. The former are represented by component middleware technologies such as CORBA and JavaBeans. They deal with external component properties (interfaces, binding mechanism, and expectations regarding the runtime environment). The second one focuses on the architecture level in which components are black-box entities. In our work, we address the integration of these two approaches.

References

1. de Paula, V. C. C.: A Formal Framework for Specifying Dynamic Distributed Architectures. PhD Thesis, Federal University of Pernambuco, Brazil, (1999)
2. Young, A. and Magee, J.: A Flexible Approach to Evolution of Reconfiguration Systems. In: Proceedings of the First International Workshop on Configurable Distributed Systems, IEE, pp. 152-163, (1992).
3. de Paula, V. C., Justo, G. R. R. and Cunha, P. R. F.: Specifying and Verifying Reconfigurable Software Architectures In: In: 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000), pp. 21-31, IEEE Computer Society, Limerick, Ireland, June (2000).
4. Spivey, J.M.: The Z Notation, A Reference Manual, Editor Prentice-Hall, (1989)
5. Oreizy, P. and Medvidovic, N. and Taylor, R. and Rosenblum, D.: Software Architecture and Component Technologies: Bridging the Gap, In: Proceedings of the OMG-DARPA Workshop on Compositional Software Architectures, Monterey, CA, January, (1998).
6. Issarny, V. and Bidan, C. and Saridakis, T.: Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype, In: Proceedings of the Fourth International Conference on Configurable Distributed Systems, pp. 207-214, Annapolis, Maryland, May, (1998).
7. Magee, J. and Kramer, J.: Dynamic Structure in Software Architectures, In: Proceedings of SIGSOFT'96 Symposium on the Foundations of Software Engineering, San Francisco, CA, October, (1996).

8. Garlan, D. and Allen, R. and Ockerbloom, J.: Architectural Mismatch or Why it's hard to build systems out of existing parts, In: Proceedings of the Seventeenth International Conference on Software Engineering, Seattle, WA, April, (1995).
9. Magee, J. and Tseng, A. and Kramer, J.: Composing Distributed Objects in CORBA, In: Third International Symposium on Autonomous Decentralized Systems - ISADS 97, pp. 9-11, Berlin, Germany, April, (1997).
10. Kramer, J. and Magee, J.: Dynamic Configuration for Distributed Systems, In: IEEE Transactions on Software Engineering, 11(4),pp.424-435, April, (1985)
11. Justo, G. R. R. and Cunha, P. R. F.: Programming Distributed Systems with Configuration Languages, In: International Workshop on Configurable Distributed Systems, London,(1992)
12. Justo, G. R. R. and Cunha, P. R. F.: An Application Framework for Dynamic Distributed Software Architectures, In: 5th International Conference on Advanced Computing (ADCOMP'97. IEEE CS Press, December, (1997)
13. Moriconi, M. and Qian, X.: Correctness and Composition of Software Architectures, In: Proceedings of ACMSIGSOFT'94: Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, 164-174, December, (1994)
14. Shaw, M. and Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, (1996)
15. Allen, R.: A Formal Approach to Software Architecture, PhD Thesis, School of Computer Science, Carnegie Mellon University, May, (1997)
16. Di Nitto, E. and Rosenblum, D.: Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures, In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los angeles, CA, USA, May, (1999)
17. da Silva, L. F. and de Paula, V. C. C.: A Meta-model to Specify Layered Software Architectures, In: Brazilian Symposium on Software Engineering (SBES'2001), October, (2001)
18. Ururahy, C; Rodriguez, N.: Alua: An event-driven communication mechanism for parallel and distributed programming. In: PDCS'99, Fort Lauderdale, Florida, (1999).
19. Ierusalimschy, R, Figueiredo, L, Celes, W.: Lua - an extensible extension language. In: Software: Practice and Experience, 26(6):635-652, (1996).
20. Batista, T. and Rodriguez, N.: Using a Scripting Language to Dynamically Interconnect Component-based Applications. To be submitted the 22th International Conference on Distributed Computing Systems (ICDCS), Viena, Austria,(2002).
21. van der Hoek, A., Heimbigner, D., Wolf, A.: Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois Technical Report CU-CS-849-98, University of Colorado, (1998).
22. Siegel, J.: CORBA: Fundamentals and Programming. John Wiley & Sons,(1996)
23. Batista, T., Chavez, C. and Rodriguez, N.: Dynamic Reconfiguration through a Generic Connector. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00), CSREA Press, Vol. II, pp 1127 – 1132, Las Vegas, USA, June (2000).
24. Batista, T. and Rodriguez, N.: Dynamic Reconfiguration of Component-based Applications. In: 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000), pp. 32-39, IEEE, Ireland, June (2000)
25. Bellissard, L. and Riveill, M.: Constructions des applications réparties, In: Ecole Placement Dynamique et Répartition de Charge, Juillet,(1996).
26. Cerqueira, R., Cassino, C., Ierusalimschy, R.: Dynamic Component Gluing Across Different Componentware Systems. In: International Symposium on Distributed Objects and Applications (DOA'99), 362-371, OMG, Scotland, September, (1999).