

# Specifying and Verifying Reconfigurable Software Architectures

Virginia C. de Paula  
Department of Informatics  
Federal University of Rio Grande do Norte  
Natal, Brazil  
vccpaula@ufrnet.br

G. R. Ribeiro Justo  
Cavendish School of Computer Science  
University of Westminster  
London, UK  
justog@cpc.wmin.ac.uk

P. R. F. Cunha  
Department of Informatics  
Federal University of Pernambuco  
Recife, Brazil  
prfc@di.ufpe.br

## Abstract

*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. Module interconnection languages (MILs) introduced the idea of creating program modules and connecting them to form larger structures. However, MILs do not support the description of important architectural elements. A new class of description languages, referred to as architectural description languages (ADLs), have recently emerged. Most ADLs, however, support only the description of static software architectures and not dynamic or reconfigurable software architectures. A further limitation of current ADLs is that they focus mainly on the formal notation and usually do not offer proof systems and tools to enable designers to formally verify the properties of their designs. We have developed the ZCL framework, which is a formal framework, specified in Z, to describe and reason about dynamic distributed software architectures. In this paper, we use a simple case study - the client-server system - to demonstrate how our formal framework ZCL can be used to specify and verify reconfigurable software architectures.*

## 1. Introduction

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 proper-

ties include the decomposition and interaction among parts as well as global system issues such as coordination, synchronization and performance [3]. Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements, the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives [21].

The idea of connecting components to form larger systems is not new. DeRemer and Kron [8] introduced the idea of creating program modules and connecting them to form larger structures separately from programming the modules themselves. The result was the first module interconnection language (MIL). Nevertheless, MILs are just a first step towards describing software structures. They allow the designer to reveal the structure of a system at the configuration level. In order to do that, typical configuration languages include the following features [4], although not all configuration languages have all features:

1. The grouping of processes into component types, and of components into further (larger) components. This feature is known as composition;
2. The parameterisation of components;
3. The instantiation of components from their types, including multiple instantiations;
4. The binding of communication among processes in one component with those in another, together with checking of the communication type;

5. The expression of an entire configuration of a distributed system as a set of such components and bindings;
6. The expression of the mapping of the software configuration onto a hardware one, and the realization of the mapping;
7. The expression of constraints so that invalid configurations and mappings can be detected; and
8. The specification of conditions for a change of configuration and the implementation of the change.

Although having a lot of benefits, MILs have some drawbacks, like failing to distinguish between implementation and interaction relationships between modules[21]. They do not support the description of important architectural elements such as a connector (which explicitly describes the interaction between the components) and architectural styles (which define generic and reusable architectural descriptions). Therefore, MILs are not suitable to deal with some architectural issues.

A new class of description languages, referred to as architectural description languages (ADLs), have recently emerged [21]. ADLs “focus on the high-level structure of the overall application rather than on the implementation details of any specific source module”[16]. Due to the novelty of the studies, there are some questions in the research community on what an ADL is and what aspects of an architecture should be modelled by an ADL. Another source of discord is the level of support an ADL should provide to developers. In [20], the authors list six classes of properties that an ADL should provide:

#### 1. Composition/Decomposition

An architectural language must allow a designer to divide a complex system hierarchically into smaller, more manageable parts, and conversely, to assemble a large system from its constituent elements.

The elements must be sufficiently independent that they can be understood in isolation from the system in which they are eventually used.

It should be possible to separate concerns of implementation level issues (such as choice of algorithms and data structures) from those of architectural structure.

#### 2. Abstraction

The architectural level of design requires a different form of abstraction to reveal high-level structure so that the distinct roles of each element in the structure are clear.

#### 3. Reusability

It should be possible to reuse components, connectors and architectural patterns in different architectural descriptions, even if they were developed outside the context of the architectural system. This form of reuse differs from the reuse of components from libraries.

#### 4. Configuration

A language for architectural description should separate the description of composite structures from the elements in those compositions. Dynamic configuration is needed to allow architectures to evolve during the execution of a system.

#### 5. Heterogeneity

There are two aspects of heterogeneity: the ability to combine different architectural patterns in a single system; and the desirability of combining components that are written in different languages. Heterogeneity is not very commonly found in the existing ADLs.

#### 6. Analysis

The need for enhanced forms of analysis are particularly important for architectural formalisms, since many of the interesting architectural properties are dynamic ones.

As stated in item 4 above, dynamic configuration is needed to allow the evolution of a system during its execution. Architectures describing these kind of systems must be represented by ADLs that permit the specification of changes. Most of existing ADLs typically support only static architecture specification and do not provide facilities for the support of dynamically changing architectures. The most common operations that can change the architecture of a system are [15]:

1. Addition of new components: it can be necessary to include new components to an architecture. The ADL must allow the inclusion of a component that was not being used before;
2. Upgrading existing components: a component can be replaced by another with the same signature (interface), but with better performance. The ideal situation is to keep the original component running, if needed, while it is being upgraded;
3. Removal of unnecessary components: if a component is no longer being used by the architecture, it can be removed;
4. Reconfiguration of application architecture: after adding or removing components, it can be necessary to reconnect components and connectors; and

5. Reconfiguration of system architecture: it can be necessary to move a component from one machine to another. The architecture must support the modification of the mapping of components to processors.

There are, however, few ADLs which support the description of dynamic architectures. Examples include C2[18], Darwin[13] and Rapide[12]. Darwin and Rapide support only constrained dynamic manipulation of architectures, where all run-time changes must be known a priori, while C2 supports pure dynamic manipulation, where no restrictions are made on the types of allowed dynamic changes at architecture specification time.

A further limitation of current ADLs is that they focus mainly on the formal notation and usually do not offer proof systems and tools to enable designers to formally verify the properties of their designs.

In this paper, we demonstrate how our formal framework ZCL[11, 6, 5] can be used to specify and verify reconfigurable systems. Section 2 gives the motivation and an overview of formal models for dynamic software architectures. In section 3, we briefly describe ZCL. In section 4, we present a case study - the client-server system - to show how ZCL can be used to specify and verify reconfigurable systems. In section 4.2, we summarise the main ideas presented in this paper and give directions for future work.

## 2 Formal Models for Software Architectures

Software architectures are usually represented informally by boxes and lines diagrams. The problem with this approach is that each author gives a different semantic to the same elements (boxes and lines) [1]. Module interconnection languages and the modularization facilities of programming languages are both unsuitable to describe software architecture, because they require the system designer to translate architectural abstractions into low-level primitives provided by the programming language. One important step towards a more scientific basis for design is an appropriate formal foundation for software architecture. Formalisms can be used to provide precise, abstract models and to provide analytical techniques based on these models. Using formal models, designers can select proper abstractions to an architectural description having the precision of a formal model. Several aspects of software architecture can be formalized [21]:

1. The architecture of a specific system: formalisms of this kind allow the software architect to plan a particular system;
2. An architectural style, which is a pattern of system structure: this kind of formalism can be used to describe architectural abstractions for families of systems;

3. A theory of software architecture: these formalisms can clarify the meaning of generic architectural concepts, such as architectural connection, hierarchical architectural representation, and architectural style and;
4. Formal semantics for architectural description languages: this kind of formalism treats architectural description as a language issue and applies traditional techniques for representing the semantics of languages.

We are interested in providing formal ADLs and theories, which enable designers to formally specify their architectures and consequently be able to prove properties of their designs. Our work provides a description, which support both static and dynamic software architecture descriptions.

As previously stated, one of the limitations of most ADLs is the lack of tools, which support for automatic analysis and proves. We overcome this limitation by providing a formal framework based on the well-known formal language Z [22, 23].

The Z language [22] is based on set theory and first order logic. It extends the use of these languages by allowing an additional mathematical type known as the **schema type**. Syntactically a schema is a box divided into two parts by a horizontal line. There are two types of schema: state and operation.

In a state schema, the upper half is known as the **declarative** part, and is used to declare variables and their types. The second part of the state schema is known as the **predicate** part, and in this part we show how the variables are related and constrained. Each schema has a distinct name. Semantically, a schema can be considered as having the same type as the *Cartesian* product of the types of its variables, without ordering, and with the state space constrained by the schema predicates. State schemas describe the possible states of a system. Modularity is facilitated in Z by allowing schemas to be included within other schemas.

Operations affect state, and are characterised by their effect on the state. An operation schema relates the state variables before and after the operation. The general operation schema has a before state (unprimed state variables), an after state (primed state variables), inputs (question-marked variables), outputs (exclamation-marked variables), and a set of preconditions for the application of the operation.

Z is an ideal means of formally presenting design structures. Through the use of an abstract specification we do not restrict a specifier to any particular implementation; rather it provides a general mathematical framework within which general concepts can be defined. In addition, Z has been successfully applied to industrial products and offers a number of supporting tools for type checking, animation and verification of specifications.

### 3. The ZCL Framework

The ZCL framework [11, 6, 5, 7] is a formal framework, specified in Z [22], to describe and reason about dynamic distributed software architectures. It focuses on the operations necessary for the construction of dynamic software architectures. Configuration and reconfiguration operations are considered. Each configuration operation has a corresponding reconfiguration one. The ZCL framework is complemented by an execution model, which is presented and formalised in [7]. So, the architect can concentrate on architectural issues or he/she can also analyse execution issues.

ZCL is based on the configuration language CL [9, 10], 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.

Static verifications can be carried automatically. For example, to verify whether a component being declared by the architecture exists in the *library of components*. The initial configuration is stored in the *configuration table*, which is dynamically modified to reflect changes suffered by the application architecture. The reconfiguration operations use the configuration table to ensure that the application is in a state suitable for modifications. All operations contain error cases also specified as schemas. When any constraint of the operation is not obeyed, the error case schema of the operation is used. The framework foresees some auxiliary operations that are used whenever necessary to guarantee the feasibility of reconfiguration operations.

#### 3.1 ZCL architectural elements

As already mentioned, ZCL was specified in Z [22], which is based on set theory and first order logic. In Z, mathematical objects and their properties are encapsulated into *schemas*: templates to declarations and constraints. A schema can be used to describe the state of a system and the different ways this state can be changed.

To construct the framework ZCL, we have considered a combination of state and operations. We modelled components, composite components, instances, ports, connections and configuration (top) in schemas separated from those of the operations.

In ZCL, an architecture has an hierarchical structure in which the configuration (architecture) is a composition of components that can also be composite. Composite components can be seen as (sub)architectures. Those components that implement a functionality are simply called components or task components. They are the smaller unit of computation we are considering. An architecture in ZCL is constructed by successive use of its operations. Nevertheless, the components must already exist in the

library of components when the creation of an architecture. ZCL includes auxiliary operations to allow the architect to add components to the library. Those operations are: *CL\_Create\_Task*, to create components; and *CL\_Create\_Group*, to create composite components, also called group components. The Z schemas that formalize those operations can be found in [7].

In the following, we present the schema which depicts the library of components. It is worth saying that a component cannot be a task one and a composite at the same time. This restriction is explicit in the schema.

$\begin{array}{l} \text{tasks} : ID\_Component \rightarrow CL\_Component \\ \text{groups} : \\ \quad ID\_Component \rightarrow CL\_Composite\_Component \\ \hline \text{dom tasks} \cap \text{dom groups} = \emptyset \end{array}$
--

Having the components in the library, the architect can use ZCL operations to create or modify an architecture. ZCL operations assure that the architecture obeys the model adopted by ZCL and that any modification will only be done when it leaves the architecture in a consistent state. The checking to guarantee the architecture consistency are performed automatically when the operations are used, since they are specified as constraints to the operations.

Figure 1 illustrate how the schemas which specify the architectural elements of ZCL are related.

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

The *CL\_Component*, shown below, represents a task component. It specifies the component interface (*interfaces*) as a set of ports (*PortNames*). It port has attributes (*Port\_Attributes*) such as: the port direction (*DIR*), to indicate if it is a port which receives (entryport) or sends (exitport) data; the port mode (*MODE*), which can be *notify*, to implement asynchronous communication, or *reply*, to synchronous communication; and the type (*TYPE*) of data supported by the port. It is also possible to specify application specific attributes by using the *component\_attr* function.

$\begin{array}{l} \text{dir} : DIR \\ \text{mode} : MODE \\ \text{type} : TYPE \end{array}$
---

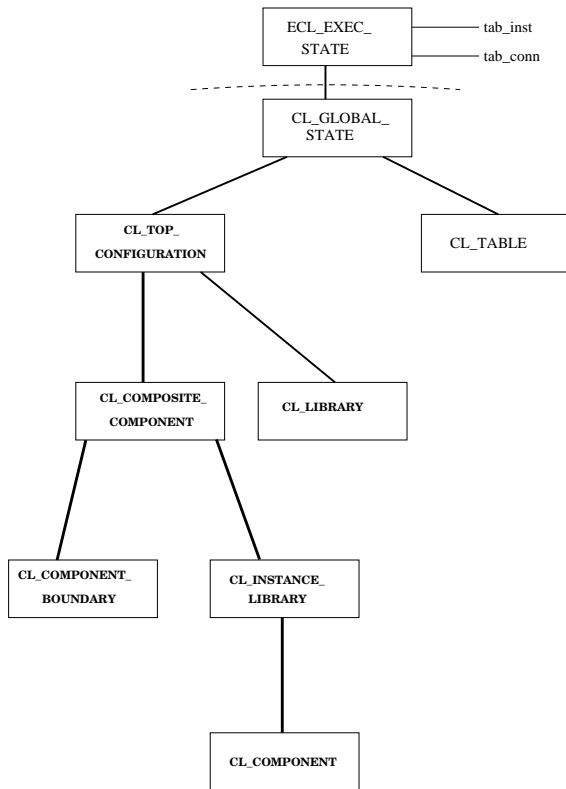


Figure 1. Overview of ZCL state schemas

$CL\_Component$ $component\_attr : Indices \rightarrow Attributes$ $interfaces : \mathbb{F}PortNames$ $port\_attr : PortNames \rightarrow Port\_Attributes$ $dom\ port\_attr \subseteq interfaces$
--

As we are worried about run-time issues, we want not only to specify an architectural component, but also to be able to represent a component that can be executed. This operational component is said to be an instance (in our specification, it is *Nodes*) and it is depicted by the *CL\_InstanceLibrary* schema. In this schema, we represent information related to all existing instances of a specific component, their location in the network, and their particular attributes by which the architect can transform the use of a component into a more specific behaviour. Moreover, we represent instances of ports which inherit port attributes. The schemas written in Z are in general too long. Although, we do not show the schemas of ZCL in this paper, the reader can find the complete specification of ZCL in [7].

A composite component is represented by the *CL\_Composite-Component*. It uses components instances to construct the structure of an application. Information concerning instance and connections are also kept, as well as those concerning sub-architectures which are part of the composite component. In ZCL, connections between ports are modeled by the *CL\_Connection* schema, which knows each pair of connected ports. A composite component has a special interface composed by *virtual ports* that are connected to the ports of its internal components.

The architecture or structure of an application is called its top configuration and is represented by the schema *CL\_Top-Configuration*. A top configuration is a composite component with no communication to other components, because it is in the higher level of the application hierarchy.

The global state of an application can be depicted considering its top configuration (*CL\_Top-Configuration*) and its configuration table (*CL\_Table*). When a component is declared to be used by an architecture, it must have been included in the library. The *InContext* set contains information about the declared components.

$CL\_Global\_State$ $CL\_Top\_Configuration$ $CL\_Table$ $(InContext \subseteq dom\ tasks \vee$ $    InContext \subseteq dom\ groups)$ $InstNodes \subseteq dom\ node\_parent$
---

### 3.2 Operations over an architecture

An architecture specification in ZCL is done by using operations which change the application global state ( $CL\_Global\_State$ ). The operations are successful when the constraints imposed by the architectural elements of ZCL are obeyed. All ZCL schemas have error cases.

To create an architecture using ZCL, we have to follow some steps:

1. To define a context ( $CL\_Define\_Context$ ): the components that the architect wants to use in the application architecture are declared and the framework verify their existence in the library. The configuration table is updated.

$CL\_Define\_Context$
$\exists CL\_Library$
$\Delta CL\_Global\_State$
$c? : ID\_Component$
$(c? \in \text{dom } tasks \vee c? \in \text{dom } groups)$
$c? \notin InContext \Rightarrow$
$InContext' = InContext \cup \{c?\}$
$c? \in \text{dom } tasks \Rightarrow$
$components' = components \cup \{c?\}$
$c? \in \text{dom } groups \Rightarrow$
$composites' = composites \cup \{c?\}$

$$CL\_Use \hat{=} (CL\_Define\_Context \wedge Success) \vee (Error\_CL\_Define\_Context \wedge Failure)$$

2. To create instances ( $CL\_Create\_Instance$ ): the operational components are created with a unique identifier and the machine in which it will be executed is informed. The component must have been declared.
3. To connect ports ( $CL\_Link$ ): communication ports are connected in order to allow the communication between instances. A  $CL\_Connection$  is created. The ports must have the same type, and compatible mode and direction.
4. To activate instances ( $CL\_Activate$ ): in ZCL, the execution of an instance does not start in the same moment it is created. The instance must be explicitly activated. So, it is possible to have a better control over parallelism between components. The configuration table is updated whenever an instance is activated.

An architect can create a static architecture using the operations described above. Nevertheless, we need operations that allow us to change existing architectures. The changes that can be carried out are:

1. To deactivate instances ( $CL\_Deactivate$ ): an instance execution can be interrupted. The configuration table must be updated.
2. To disconnect ports ( $CL\_Unlink$ ): the disconnection of two ports can occur when they are not changing messages. Messages not consumed are lost. The corresponding  $CL\_Connection$  must be destroyed.
3. To relink ports ( $CL\_Relink$ ): in this operations, two connected ports are disconnected, but messages not read (consumed) can be transferred to a new connection in which one of the just disconnected ports is now involved.
4. To delete instances ( $CL\_Delete\_Instance$ ): an instance can be eliminated from an architecture. It must not be activated or have connected ports.
5. To remove a component from the context ( $CL\_Remove\_Component$ ): when a component is not been used anymore, it can be removed from the context of an architecture. It must not have instances and new instances of that component can not be created.

Those alterations can occur when the application is in execution. Therefore, the framework must guarantee that the framework will be in a consistent state and that the execution can continue after the changes are done. In this section, we have mentioned the operations used to change an architectural description. In the following, we explain how the execution level deals with those changes and how the framework can check the application state.

### 3.3 The ZCL execution model

In order to analyse how changes at the architectural level affect the execution of an application, we have considered instances and connections as state machines by which we can know the state of an instance or of a connection when needed. Each ZCL operation previously described, has a new corresponding operation, also specified by schemas in Z, which changes the state of instances and/or connections. The architect can choose between using just the operations over architectural elements or invoking the operations that deal with execution states. Z allows us to compose schemas. So, different schemas can be invoked together. In order to illustrate schemas composition, we show below the composition of the operations to activate instances. The configuration table is updated, as well as, the instance state. The framework checks the instance state, which must be in the passive state to be activated. All schemas specifying operations at the execution model level are composed by a schema to check the state at the moment and by another

schema to change the state of the instance or connection if it is in an appropriate state to be changed. In the case of the activation of instances, we have respectively the schemas  $ECL\_Activate\_Cond$  and  $ECL\_Activate\_Op$ .

$$\frac{ECL\_Activate\_Cond}{\begin{array}{l} \exists ECL\_Exec\_State \\ node? : Nodes \\ \hline (node?, passive) \in tab\_inst \end{array}}$$

$$\frac{ECL\_Activate\_Op}{\begin{array}{l} \Delta ECL\_Exec\_State \\ node? : Nodes \\ \hline tab\_inst' = tab\_inst \oplus \{node? \mapsto processing\} \\ tab\_conn' = tab\_conn \end{array}}$$

$$\begin{aligned} ECL\_Start &\hat{=} (ECL\_Activate\_Cond \wedge Success) \\ &\quad \wedge ECL\_Activate\_Op \\ ECL\_Activate &\hat{=} CL\_Activate \circ ECL\_Start \end{aligned}$$

The  $ECL\_Exec\_State$  schema represents the state of an application, including its architectural description ( $CL\_Global\_State$ ) and the states of instances ( $tab\_inst$ ) and connections ( $tab\_conn$ ) at execution time. The relationship between  $ECL\_Exec\_State$  and the architectural elements of ZCL is shown in Figure 1.

$$\frac{ECL\_Exec\_State}{\begin{array}{l} CL\_Global\_State \\ tab\_inst : Nodes \rightarrow Inst\_State \\ tab\_conn : CL\_Connection \rightarrow Conn\_State \\ \hline \text{dom } tab\_inst \subseteq \text{dom } node\_parent \\ \text{dom } tab\_conn \subseteq connection \end{array}}$$

In summary, each operation used by an architect at the architectural level has a corresponding one at the execution model level, to change the states of instance and connections according to the operation performed. The framework always checks conditions to guarantee that the constraints are satisfied in order to permit the changes required. The state diagrams that include all sub-states an instance can assume as well as, those related to connections, can be found in [7].

Besides knowing the state of each instance and connection, the framework ZCL must know the instances behaviour to decide if a change in an architecture (reconfiguration) can be performed. We define the behaviour of an

instance in a very simple way: it can be communicating or computing (processing). We said it is computing whenever it is not communicating. The specification of the  $ZCL\_execution\_machine$ , which controls the behaviour of instances, is given by an executor ( $ECL\_Component\_Executor$ ) and a dispatcher ( $ECL\_Executor\_Dispatcher$ ). They have the responsibility for controlling instances execution. The executor knows the state of each instance. So, it knows which of them are being executed or are blocked, for example. The dispatcher is the one that decide which instance, from those which are waiting to be executed, will be the next one to be executed. We have also specified the communication system in Z. Therefore, the architect can know the state of connections to decide if a reconfiguration can be carried out. The communication system is always used when a communication command (send or receive) is found in an instance behaviour. ZCL also specifies some operations to allow the architect deal with execution queues in order to analyse the dynamic behaviour of an application when a reconfiguration is requested.

As said in [17], 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 construct 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 separate the schemas related to structural (static) analysis from those related to dynamic analysis. This means that the execution model can be easily replaced or modified. In Figure 1, schemas named with the prefix  $CL\_$  are related to architectural concepts and those named with the prefix  $ECL\_$  are related to run-time concepts.

The ZCL framework enables us to perform some analysis on the specification. In the case of reconfigurations, it is possible to evaluate what type of pre-defined conditions can be included in a configuration and their effect on the consistency of the configuration. Another type of analysis that can be done is to establish invariant constraints for the configuration and evaluate how they effect the reconfigurability of the configuration.

In summary, ZCL is a formal framework which enables designers to hierarchically specify the distributed software architecture of their systems, including reconfiguration properties, and validate the architecture. In the next section, we illustrate the use of ZCL in specifying and verifying a reconfigurable client and server application.

## 4 The Client-Server System

This section presents a simple case study, which illustrates the key aspects of ZCL in the development of re-

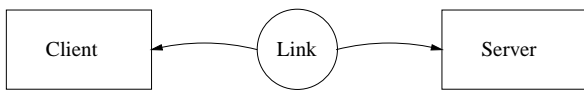


Figure 2. Client-Server System



Figure 3. Client-Server System using ZCL

configurable applications. The selected case study first appeared in [2], where the authors have defined a reconfigurable model for the Wright ADL. The reader will observe that the solution based on ZCL is better structured as a consequence of the model being originally conceived to support reconfiguration. In addition, designers can use Z tools such as Z-Eves, as illustrated later in this section, to simulate and verify their design.

The architecture of the Client-Server System, presented in [2], is illustrated by a simple diagram, showing Client components linked to a Server component (Figure 2). There must exist at least one Client component and only one Server. Each Client must be connected to the Server. These are important constraints of this application.

Using the concepts of the ZCL, the architecture of the Client-Server system can be represented by Figure 3, where one instance of client is connected to the instance of server through a communication port.

We can use the CL language[9, 10] as a notation to show the steps to be followed to create an initial architecture of the client-server system in ZCL:

```

system ClientServer;
begin
/* define context: component types */
  use task Client;
  use task Server;
/* create component instances */
  create srv from Server;
  create clt1 from Client;
/* link instances' ports */
  link clt1.require to srv.provide;
/* start instance execution */
  activate srv, clt1;
end;

```

## 4.1. The Client-Server System in ZCL

The software architecture of an application is specified in ZCL by instantiating the operation schemas and defining values for the required parameters.

As previously said, in ZCL, ports must have a direction type: exit or entry; and a transaction mode: notify (one-way) or requery (two-way). In the case of the Client-server system, we assume that the client requires an exit port and the server provides an entry port. The ports define a two-way transaction. We create a new task component using the auxiliary operation *CL\_Create\_Task*. Each port of the component's interface must be created using the *CL\_Create\_Port* schema. After creating a new component and its interface, we use the *CL\_Update\_LibSimple* schema to include the new component in the Library. This schema receives the identifier of the component as parameter and the library is responsible for binding identifiers and components as the reader can see in the *CL\_Library* schema presented in section 3.1.

$$\begin{aligned}
 &CL\_Create\_Task[comp\_attr? := \emptyset, \\
 &\quad ports? := \{provide\}] \\
 &\wedge CL\_Create\_Port[port? := provide, d? := entry, \\
 &\quad m? := requery, t? := string] \\
 &\wedge CL\_Update\_LibSimple[nc? := Server]
 \end{aligned}$$

As previously stated, Z is based on set theory, therefore after each operation the sets are updated. For example, after performing the above operation to include the component *Server* in the ZCL library, the set of tasks, which stores the information about the components and their interfaces, will be updated.

In the case where the component has more than one port in the interface, it is necessary to use the *CL\_Create\_Port* for each port in the interface.

The architecture description of an application is represented by the *CL\_Top\_Configuration* schema. After including the necessary components in the library, we use the ZCL operations to create the desired architecture step-by-step. Remember that each ZCL operation may contain constraints (preconditions) which must be checked before the operation is executed. In addition, the designer can specify application-specific constraints. For example, in the case of the Client-Server application it is required that there must exist at least one active server and one client during the whole execution. This constraint is specified in the *ExampleCS* schema, where an instance of the component server must exist, not be in the unborn state (an instance is in the unborn state when deleted) and belong to the set of active instances. The *ExampleCS* schema also states that

the instances of *Client* must be connected to the instance of *Server*.

<i>ExampleCS</i>
<i>ECL_Exec_State</i>
<i>CL_Connection</i>
$\begin{aligned} &\#children(Server) \geq 1 \\ &\#children(Client) = 1 \\ &\exists Srv : Nodes \mid Srv \in children(Server) \bullet \\ &\quad Srv \in ActiveInst \wedge tab\_inst(Srv) \neq unborn \wedge \\ &(\forall Clt : Nodes \mid Clt \in children(Client) \\ &\quad \bullet \exists conn : CL\_Connection \bullet \\ &\quad \quad conn = \theta CL\_Connection \\ &\quad \quad [receiver := (Srv, provide), \\ &\quad \quad \quad sender := (Clt, require)] \\ &\quad \wedge conn \in connection) \end{aligned}$

For completeness all schemas have error cases where the preconditions are not satisfied. So the complete set of constraints for the Client-Server application is specified by a disjunction of the above schema and the error schema:

$$CS \hat{=} \langle ExampleCS \wedge Success \rangle \vee \langle Not\_ExampleCS \wedge Failure \rangle$$

In ZCL, we can verify application specific constraints and analyse various run-time properties of the architecture. We have used Z-Eves [19], a theorem prover for Z, to simulate and verify an architecture. Z-Eves allows Z specifications to be analysed in a number of different ways. Using Z-Eves the state of the architecture, as defined by the sets specified in state schemas, can be inspected and modified after each operation.

In [2], the authors suggest three ways to simulate dynamism in Wright in order to allow the reconfiguration of the Client-Server system in case of fault. In the first proposed solution, they consider two servers: a *primary* server, which is more desirable to use, but which may go down unexpectedly, and a *secondary* server, which, while reliable, provides a lesser form of the service [2]. In this approach, there are two instances of the *Server* component. Nevertheless, just one is activated. When the *primary* instance is not active, the *secondary* must be activated automatically. The problem with this solution is when both instances go down, because the instances have been previously created and there is no possibility of creating a new one dynamically.

The second approach suggested in [2] has two configurations, each simple in itself, which alternate as the primary server goes down and comes back up, in order to provide constructs to describe the dynamics of the system explicitly.

Wright does not have a notation for characterizing changes in the architecture during a computation. Therefore, a new notation should be introduced. Such a characterization includes [2]:

1. What events in the computation trigger a reconfiguration, and
2. how the system should be reconfigured in response to a trigger

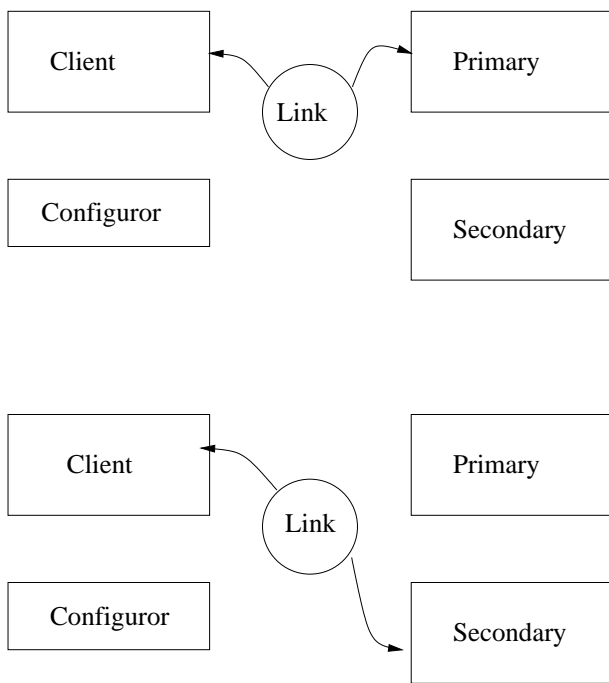
In ZCL, we can specify situations in which a reconfiguration can take place. We can then reconfigure the system maintaining its constraints using the ZCL operations. Reconfigurations in ZCL can be done at two levels: the architectural level, involving just issues related to the design level; and the execution level, including run-time issues that can cause a reconfiguration. In this way, ZCL is a very useful tool to allow the designer to simulate some real run-time situations.

We adopted the third approach presented in [2] to allow the solution of the system in Wright. In this approach, special *control* events are introduced into a component's alphabet, and allowed to occur in port descriptions. Therefore, the interface of a component was extended to describe when reconfigurations were permitted in each protocol in which it participates (Figure 4). These control events are used in a separate view of the architecture, the configuration program, which describes how these events trigger reconfigurations.

The control events mentioned by [2] corresponds to the configuration program (architecture description) supported by ZCL. A reconfiguration can be planned in such way that the system always keeps its constraints. We can determine the creation of a new instance of a *Server* when the current one is not running and ensure there is just one *Server* activated at some moment. In our example, we assume the *Client* behaviour to be defined by performing internal computations (process action) between sending and receiving a message from the *Server*. The behavior of the *Server* is also defined by performing internal computations when not communicating with the client. The behaviour of the components are specified in ZCL as follows:

$$\begin{aligned} inst\_action ::= &\{ (clt \mapsto \\ &\quad \langle process, send, process, recreply, process \rangle), \\ & (srv \mapsto \\ &\quad \langle receive, receive, process, process, receive \rangle) \} \end{aligned}$$

where *clt* and *srv* are instances of *Client* and *Server*; *receive* says that the component is now ready to receive data by one of its ports; *sends* says that the component will



**Figure 4. Alternating Configurations of Client-Server System**

send data by one of its ports; *recreply* says that the component is waiting for a reply; and *process* indicates that the component is not communicating.

Let's now assume that our application has one instance of Server, *srv*, and two instances of Client, *clt1* and *clt2*. Although the instance of Server is active, its state is *unborn* for some reason. So, our application is not in a consistent state, because it is not obeyed the constraints imposed to it.

As previously described, the *CS* schema is a combination of *ExampleCS* schema and *Not.ExampleCS*, which has as predicates the negation of those of *ExampleCS*. The *CS* schema produces as output the *r!* variable with value *success* when the first schema is true or *failure* otherwise. When the instance of Server is *unborn* (Remember that in ZCL, the state of an instance turn to *unborn* when for some reason the instance is deleted.), the simulation will return an error in the application, i.e. a precondition of *ExampleCS* is not true. The precondition of an operation schema describes the set of states for which the outcome of the operation is properly defined [23]. When an instance is in the *unborn* state, it is necessary to reconfigure our application to have a new instance of Server and to obey its the constraints. That can be done step-by-step or creating a schema to do that automatically when the error is detected. Below we specify a sequence of operations using the Z operator  $\circ$  to combine schemas to create a new instance of

Server (*ECL.Create.Instance*), to connect that instance to the existing instances of Client (*ECL.Link*) and finally to activate the new instance (*ECL.Activate*).

```

NewServer  $\hat{=}$ 
  ECL.Create.Instance[node? := newsrv,
    component? := Server, machine? := maq1]  $\circ$ 
  ECL.Link[sender? := (clt1, require),
    receiver? := (newsrv, provide)]  $\circ$ 
  ECL.Link[sender? := (clt2, require),
    receiver? := (newsrv, provide)]  $\circ$ 
  ECL.Activate[node? := newsrv]

```

As previously stated, the state sets can be inspected during the simulation (we have used Z-Eves). For example, it is possible to detect when the active instances of Client are connected to a Server by examining the existence of the connection. If the connection does not exist, the corresponding ZCL operation can be used to reconfigure the system and to connect the instances. More complex analysis is also possible. For example, the state of an instance or of a connection can be tested to verify whether the configuration is in a valid state in order to the reconfiguration to be carried out.

## 4.2. Conclusions

ADLs offer a new direction to the development of complex distributed component-based systems, as the design structure can be clearly specified. This enables designers to check various architectural properties such as scalability and use of resources. It can also provide important foundation for reusability, evolution and run-time reconfiguration. Most of the existing ADLs, however, do not support specification of reconfigurable architectures. Darwin [14] is one of the exceptions. Although the work enables us to verify some properties of (dynamic) configurations described in Darwin, it is more concerned with proving the correctness of the Darwin elaboration mechanism, namely, that after Darwin transforms a hierarchical configuration into a flat-ten configuration, it preserves the correctness of the modules and their interconnections.

In this paper, we have shown that ZCL is useful in specifying and verifying reconfigurable systems. By using a formal language we can formally analyse and prove properties of a configuration and their changes. Our framework deals with operational aspects as to how the reconfiguration takes place and provides a powerful method to verify properties of the configuration. In addition, a designer can easily specify a system in ZCL just being familiarized with the operations of the framework and he/she can use a theorem prover, as we have used Z-Eves, to verify properties of the

specified system. We have already developed some applications using ZCL [5, 7] but in order to apply it to industrial cases we are developing a user-friendly graphical interface, where designers will not have to directly manipulate the Z specifications.

## References

- [1] G. Abowd, R. Allen, and D. Garlan. Using Style to Understand Descriptions of Software Architecture. *ACM Software Engineering Notes*, 18(5), December 1993.
- [2] R. Allen, R. Douence, and D. Garlan. Specifying and Analysing Dynamic Software Architectures. *Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal*, March 1998.
- [3] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.
- [4] J. M. Bishop. Languages for Configuration Programming: a comparison. Technical report, University of Pretoria, South Africa, 1994.
- [5] V. C. de Paula, G. R. Justo, and P. Cunha. Formal Specification of Dynamic Architectural Styles. In *1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, USA*, 1999.
- [6] V. C. de Paula, G. R. R. Justo, and P. R. F. Cunha. Specifying Dynamic Distributed Software Architectures. *Proceedings of the XII Software Engineering Brazilian Symposium*, pages 7–22, October 1998.
- [7] V. C. C. de Paula. *ZCL: A Formal Framework for Specifying Dynamic Distributed Software Architectures*. PhD thesis, Department of Informatics, Federal University of Pernambuco, Recife, Brazil, 1999.
- [8] F. DeRemer and H. H. Kron. Programming-in-the-large versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.
- [9] G. R. R. Justo and P. R. F. Cunha. Programming Distributed Systems with Configuration Languages. *International Workshop on Configurable Distributed Systems. IEE, London, UK*, pages 118–127, 1992.
- [10] G. R. R. Justo and P. R. F. Cunha. An Application Framework for Dynamic Distributed Software Architectures. In *5th International Conference on Advanced Computing (ADCOMP'97. IEEE CS Press.*, December 1997.
- [11] G. R. R. Justo, V. C. de Paula, and P. R. F. Cunha. Formal Specification of Evolving Distributed Software Architectures. *Proceedings of Ninth Workshop on Database and Expert Systems Applications (DEXA'98), International Workshop on Coordination Technologies for Information Systems (CTIS'98), Vienna, Austria*, pages 548–553, August 1998.
- [12] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.
- [13] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. *Fifth European Software Engineering Conference (ESEC'95)*, September 1995.
- [14] J. Magee, N. Dulay, and J. Kramer. A Constructive Development Environment for Parallel and Distributed Programs. *2nd International Workshop on Configurable Distributed Systems, SEI, Carnegie Mellon University. IEEE Computer Society Press.*, March 1994.
- [15] N. Medvidovic. ADLs and Dynamic Architecture Changes. *Proceedings of the Second International Software Architecture Workshop (ISAW-2), San Francisco, CA, USA*, pages 24–27, October 14–15 1996.
- [16] N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical report, Department of Information and Computer Science, University of California, Irvine, USA. UCI-ICS-97-02, February 1997.
- [17] M. Moriconi and X. Qian. Correctness and Composition of Software Architectures. *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA*, pages 164–174, December 1994.
- [18] P. Oreizy. Issues in the Runtime Modification of Software Architectures. Technical report, Department of Information and Computer Science, University of California, Irvine, USA. UCI-ICS-TR-96-35, August 1996.
- [19] M. Saaltink. The Z/EVES System. In *ZUM'97: The Formal Specification Notation (10th International Conference of Z Users), Reading, UK*, April 1997.
- [20] M. Shaw and D. Garlan. Characteristics of Higher-level Languages for Software Architecture. Technical report, School of Computer Science, Carnegie Mellon University, USA. CMU-CS-94-210, December 1994.
- [21] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [22] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall, 1989.
- [23] J. Woodcock and J. Davies. *Using Z - Specification, Refinement, and Proof*. Prentice Hall, 1996.