Specifying Dynamic Distributed Software Architectures

Virginia C. de Paula[1] and *G. R. Ribeiro Justo*
Centre for Parallel Computing, University of Westminster, London, UK
{vccpaula,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 is important to the design of complex software systems, as it provides a model of the large scale structural properties of the system. It is possible to find several formal models to depict static distributed software architecture. Nevertheless, notations for supporting architectural dynamism and evolution are still difficult to find in the literature. We present a formal framework to specify dynamic distributed applications to enable us to specify the dynamic behaviour of reconfigurable systems. This framework will help the designer to check the project suitability.

1 Introduction

The software architecture of a system is important to the design of complex software systems, as it provides 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 [5]. Structural issues also 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 [29].

Research on distributed software architectures evolved from the existing MILs[29], especially configuration languages[6, 9, 14], which basically separate the computation from the structure of the system, defining a special notation to describe the architectural elements of a distributed system. Despite of their benefits, MILs have some drawbacks, like failing to distinguish between implementation and interaction relationships between modules[29]. Therefore, MILs are not suitable to deal with some architectural issues.

In order to deal with those architectural issues, architectural description languages (ADLs) have emerged as an important field of study. Most of existing ADLs typically support only static architecture specification and do not provide facilities for the support of dynamic architectures, used to describe architectures which change during run-time. The most common operations that can change the architecture of a system are [16]: addition of new components, upgrading existing components, removal of unnecessary components, reconfiguration of application architecture, and reconfiguration of system architecture. Notations for supporting architectural dynamism and evolution is still difficult to find in the literature. Exceptions are C2[22], Darwin[15] and Rapide[13].

[1]On leave from Federal University of Pernambuco (PhD student) and from Federal University of Rio Grande do Norte (Assistant lecturer).

In this paper, we present ZCL: a formal framework, specified in Z [30], to specify and reason about dynamic distributed software architectures. The framework is based on the CL model [9, 10] and focuses on the operations necessary in the construction of dynamic software architectures. The ZCL framework enables us to perform some analysis on the specification of an architecture, as we present on Section 5. The framework has also an execution machine, which is still being formalized, which will allow us to analyse the behaviour of an (dynamic) architecture during run-time.

In section 2, we present the features one language must have to be considered an ADL, including dynamism. In section 3, we present some formal models for architectural descriptions. In section 4, we present the ZCL framework. In the sequence of the paper, we present, in section 5, a brief case study to show how the framework can be used. In section 6, we describe the next steps of our research, including the reconfiguration model for ZCL, and in section 7 we conclude the paper emphasizing our contribution.

2 ADLs

In this section, we present the features one language should have to be considered an ADL. We give special attention to ADLs that support the specification of dynamic architectures.

ADLs "focus on the high-level structure of the overall application rather then the implementation details of any specific source module"[17]. 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 disagreement is the level of support an ADL should provide to developers. In [28], the authors list six classes of properties that an ADL should provide:

1. Composition: 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 as to allow them to 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 structures 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. This property 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.

The survey presented by Medvidovic in [17] compares the most popular ADLs: Aesop, C2, Darwin, MetaH, Rapide, SADL, UniCon, and Wright. ACME [8] is a common denominator of existing ADLs, providing a fixed vocabulary for representing architectural structures and an open semantic framework in which architectural structures can be annotated with ADL-specific properties.

2.1 ADLs and Dynamic Architectures

Architectures are likely to describe large, long-lived software systems that may evolve over time. ADLs must support such changes through features for modelling evolution (before execution) and dynamism (during execution). This is done at the level of configurations.

As we previously said, the most common operations that can change the architecture of a system are [16]:

1. Addition of new components: it can be necessary to include new components to an architecture. So, 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 same functionality, but with better performance, for example. 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 more 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. It this case, the architecture must support the modification of the mapping of components to processors.

C2, Darwin, and Rapide support dynamism. Darwin and Rapide support only constrained dynamic manipulation of architecture, i.e. the changes must be planned.

In [22] and [23], we find a set of issues to be considered when trying to establish under what circumstances it is safe to remove and/or add a component from/to an architecture, change the filtering policy on a connector port, and rewire the architecture.

3 Formalizing Software Architectures

Several formalisms are being investigated by the software architecture research community. The dimension of generality and power of the formalism must be considered. A formal notation for architectural descriptions might be useful both for system construction as well as support verification. In most cases, however, a formalism has at its core a specific problem that it is attempting to address[27]. Here are four core functions in this design space:

- Analysis of Architectural Instances

  To analyse designs, it is necessary to associate an underlying semantics model with the description of a system architecture. Several different models have been proposed, each of them focusing on one important aspect. UniCon and Aesop, for example, support methods of real-time analysis, while Darwin [15, 24] allows dynamic architectures modelling systems behavior in terms the of π-calculus.

- Capture of Architectural Styles

  Some styles have been completely formalized. The framework developed by Abowd, Allen, and Garlan [1], for example, permit the comparison of different styles at a semantic level.

- Verification of Architectural Styles

  Sometimes architectural descriptions must be refined into lower-level architectural descriptions that are more directly implemented than their abstract counterparts. In [18], the authors observed that it is possible to exploit patterns of refinement between different levels of architectural description. SADL [19] is an ADL constructed with the possibility of refining architectural descriptions.

- Analysis of Architecture in General

  It is important to provide a formal basis of an architectural description in order to answer some questions that arise. Some languages define connectors as first-class entities or allow the specification of components or protocols to reason about these elements, guaranteeing, for example, that components interacting over a given connector will never deadlock.

The model defined by Abowd, Allen and Garlan[1] provides a formal framework for the uniform definition of architectural styles. In order to do that, architectural styles are described formally in terms of a small set of semantic mappings. The model shows how these mappings can be used to define formally two common architectural styles. In this way, new styles can be defined by a similar set of definitions and it is possible to use the formal descriptions to gain insight into the properties of a style and its relationships to other styles. The main feature of this model is that it gives meaning to architectural descriptions.

The Module Interconnection Language (MIL) model was first defined in [25] and incorporates the essential elements and operations of MILs as generic Z schemas. The basic elements of a system structure (or configuration) are the *Templates* (which correspond to modules) and their interfaces (*Ports*). It is assumed that there is a set of *Templates*

and a set *Ports*. The *Primitives* are a set of *Templates* that are initially available. Each port must be associated with a template and with a set of attributes (*Attributes*). Each attribute of a port is specified by a field, an element of the set of *Indices*. The primitive templates, interfaces, and attributes constitute the *library* (MIL_Library). To construct a system structure, each library template can be instantiated as a *node* which is an element of the set *Nodes*. A function *node_parent* specifies the relationship between nodes and templates. Each instantiated node inherits the interface (ports) of its parent template. However, the ports associated with the parent templates are also instantiated as *slots* associated with the node instance. Therefore, the slots must inherit the attributes of the parent port as the third constraint of the *MIL_Setting* schema indicates. *Labels* are also associated with slots of the nodes to define a relation (connection) on the set of nodes. Two nodes are considered connected if they have slots with the same label.

The Architectural Style Description Languages (ASDL) model, described in [26], is in fact an extension of the MIL model. To enable the description of architectural styles, the ASDL model mainly introduces the notion of semantics for both components and interactions. Although ASDL provides support for describing the essential elements of software architectures, including the semantic aspects of the modules and their interfaces, it does not provide support for the description of evolving (reconfigurable) software architectures.

Wright [5] is an architectural description language based on the formal description of the abstract behavior of architectural components and connectors. Wright provides a formal basis for the description of both architectural configurations and of architectural styles. It is distinguished by the use of explicit, independent connector types as interaction patterns, the ability to describe the abstract behavior of components using a CSP-like notation, the characterization of styles using predicates over system instances, and a collection of static checks to determine the consistency and completeness of an architectural specification. As the semantics of Wright specifications are formally defined, an architecture characterized in Wright provides a sound basis for reasoning about the properties of the system or style described.

Darwin has been formally specified in [15], where we find the description of the operational semantics of Darwin in terms of the π-calculus. The model is used to argue the correctness of the Darwin elaboration process and the objective is to provide a soundly based notation for specifying and constructing distributed software architectures. In [24], the Darwin's representation in first order logic is presented. The definitions and axioms of the logic representation form a theory. From this theory it is possible to derive notations of validity for both programs and configurations as well as an important property of Darwin programs - a running configuration can be extended without requiring reconfiguration. The basic features of Darwin, concerned with binding, instantiation and hierarchy, and their semantics in the π-calculus are found in [15]. Nevertheless, Darwin also has the ability to specify architectures which change at run-time using lazy and direct dynamic instantiation. So, the authors present an extension of the π-calculus model to describe the direct dynamic instantiation facility.

The problem of capturing dynamic architectures is addressed in [3]. Dynamic systems are defined as systems in which composition of interacting components changes during the course of a single computation. This is different from *steady-state behavior*, in which the computation performed has no reconfiguration. The approach of this model is based on the premise that it is both possible and valuable to separate the dynamic re-configuration behavior of an architecture from its non-reconfiguration functionality. While Darwin

capture reconfiguration behavior, at the architectural level, it is important to provide a notation that supports both aspects of design while maintaining a separation of concerns [4]. The model is then a new technique by which these two aspects can be described in a single formalism while keeping them as separate views. Analysis of the combined interaction between the two is supported.

4 The ZCL Framework

We propose a framework to help designers in the specification of an application. The main inspiration for our work is CL [9, 12]. The CL model follows the principles of others MILs but it had introduced new concepts such as the notion of *planned* reconfiguration, which is a modification considered by the designer as possible to happen.

Our framework [20, 11] is specified in Z [30] and defines a semantics to CL. To construct the framework, we have considered the CL language as a combination of state and operations. So, we modeled components, composite components, instances, ports, connectors and configuration (top) in schemas separated from the ones of the operations. This structure follows the types of schemas defined in Z, in which there are two types of schemas: state and operation. In a state schema, the upper half is know 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 it is described how variables are related and constrained. Operations affect states, and are characterized 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, an after state, inputs, outputs, and set of pre-conditions for the application of the operations.

4.1 The Software Architecture State

The state of a software architecture is divided into two parts: static and dynamic. The static state relates to the various levels of the library of components. The dynamic state relates to information about the execution.

4.1.1 The Basic Library of Components and Ports

The basic elements of a ZCL specification are *Components*, *Ports* and *Connectors*. Components can be primitive components (tasks) or composites components (group or sub-configuration). Ports can be simple ports or family of ports.

We use *Indices* and *Attributes* to classify the attributes of ports and components. Like in [25], *Attributes* are classified into fields, which are elements of the set of *Indices*. For example, in a ZCL specification, the attribute $mode ::= \{notify, reqreply\}$ are used to represent the mode of a port. A *notify* port implements asynchronous communication and a *reqreply* (request-reply) port is the one which implements synchronous communication.

The *CL_Component* schema, shown in Figure 1, specifies the interfaces as a mapping between components and sets of Ports, and defines the attributes of each port. The schema asserts that the interfaces use distinct ports (*setdisjoint* interfaces) and that every port has a direction to indicate that it receives messages (entryport) or sends messages (exitport); and has a mode to indicate that it can be *notify* or *reqreply*.

$[X]$

setdisjoint _ : $\mathbb{P}(\mathbb{P}(\mathbb{P}X))$

$\forall xss : \mathbb{P}(\mathbb{P}X) \bullet$ setdisjoint (xss)
$\Leftrightarrow (\forall xs, ys : \mathbb{P}X \bullet ((xs \in xss) \wedge (ys \in xss) \wedge (xs \neq ys)) \Rightarrow (xs \cap ys) = \emptyset)$

CL_Component[*Indices*, *Attributes*]

id_component : *Names*
component_attr : *Component* $\nrightarrow$ *Indices* $\rightarrow$ *Attributes*
interfaces : *Component* $\rightarrowtail$ $\mathbb{F}_1$ *Ports*
port_attr : *Ports* $\nrightarrow$ *Indices* $\rightarrow$ *Attributes*

setdisjoint {interfaces}
dom *port_attr* = $\bigcup$(ran *interfaces*)
$\forall p$: *Ports* | $p \in$ dom *port_attr*$\bullet$
port_attr$(p)(dir) \in \{entry, exit\} \wedge$
port_attr$(p)(mode) \in \{notify, reqreply\}$

Figure 1: **CL_Component**

4.1.2 Basic Software Architectures and Composite Components

A basic software architecture is a component that can be composed by others components (task or group) and is represented in ZCL as a composite component. In Figure 2, we present the *CL_Composite_Component* schema just to illustrate the specification. The complete framework can be found in [21].

The *Nodes* given set represents the concept of instance of a component (*CL_InstanceLibrary*). Components are instantiated into nodes and their ports into *port_inst* to form a composite component. The variable *node_parent* indicates the instantiation of components and *port_inst* the instantiation of ports respectively. The function *childrens* provides the set of nodes of one component and *node_attr* is the set of attributes of a node. In this set, the location of the node is stored.

The composite component has *virtual_ports*[2] in its interface, which can be bound to its component's interfaces. A composite component uses instances of components to construct the structure of an application. It keeps information about its *components* and the links between them (*connection* and *cname*). The variable *composites* stores information about sub-composite components.

CL_Connector stores information about each pair of ports connected, creating links. Its semantics can specify several forms of communication (*ConnectorDescriptions*).

The highest component in the hierarchy is called the Top Configuration. It is a composite one, but has no interface, as it cannot be instantiated and connected to other components.

[2]The idea of virtual_ports is similar to that presented in [26].

$[SemanticDescriptions]$

$$
\begin{array}{|l}
\hline
CL\_Component\_Boundary[Indices, Attributes] \\
interface\_attr : Ports \nrightarrow Indices \rightarrow Attributes \\
virtual\_ports : \mathbb{F} Ports \\
\hline
virtual\_ports = \text{dom } interface\_attr \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
CL\_Composite\_Component[Indices, Attributes, SemanticDescriptions] \\
CL\_Component[Indices, Attributes] \\
CL\_Component\_Boundary[Indices, Attributes] \\
CL\_Connector \\
\Xi CL\_InstanceLibrary \\
id\_composite : Names \\
components : \mathbb{F} CL\_Component \\
composites : \mathbb{F} Component \\
bind : (Nodes \times Ports) \nrightarrow \mathbb{F} Ports \\
virtual\_port\_descr : Ports \nrightarrow SemanticDescriptions \\
connection : (Nodes \times Ports) \nrightarrow ConnectorNames \\
cname : ConnectorNames \rightarrow CL\_Connector \\
\hline
\forall c : Component \mid c \in composites \bullet c \in \text{ran } group \\
\text{dom } cname = \text{ran } connection \\
\forall n : Nodes;\ p : Ports \mid (n, p) \in \text{dom } connection \bullet (n, p) \in port\_inst \\
\text{dom } bind \subseteq \text{dom } connection \\
\bigcup(\text{ran } bind) \subseteq virtual\_ports \\
\text{dom } virtual\_port\_descr = virtual\_ports \\
\forall p : Ports;\ n : Nodes \mid p \in virtual\_ports \wedge p \in bind(n, p) \bullet \\
\quad interface\_attr(p) = port\_attr(p) \\
\forall n : Nodes;\ p : Ports \mid node\_parent(n) \in composites \wedge \\
\quad p \in interfaces(node\_parent(n)) \bullet \\
\quad (n, p) \in \text{dom } bind \\
\hline
\end{array}
$$

Figure 2: **CL_Composite_Component**

A configuration table stores information about the configuration context, instances and their status, and links. The state of an architecture is important to guarantee the integrity of the application during reconfiguration. We create a schema that describes sets that keeps information about the existing elements of an architecture.

4.2 The Operations on a Software Architecture

The dynamic ZCL operations are applied at the elements of the architecture. However, it can be necessary to use some auxiliar (basic) operations applied to lower level concepts. For example, if a component is used by an application, if must exist in the library. If it does not exist in the library, it must be created. The creation of a component is not an ZCL operation, but is is necessary to define a component in a system context. Dynamic operations usually change the state of the software architecture. All operations contain error cases.

We describe below the steps necessary to create an application using the ZCL framework.

1. The first step to allow operation in an architecture is to create a *system* (application), which is named *CL_Top_Configuration*. The configuration table of the system is initialized.

2. The definition of context consists mainly in selecting components from the library and updating the configuration table. A component must be included in the context before an instance of it is created. A component can exist in the library or can be included in it. In the first case, as specified by the *CL_Define_Context* schema, the component must be declared just once. If the component does not exist in the library, it must be created (*CL_Create_Component*).

3. When a component is created, its interface is also created and the attributes of each port are defined.

4. Connectors must be created to allow the communication of the instances. Each connector has an identifier and a behaviour that must be defined by the architect. Each connector can support more than one pair of communicating ports.

Instances of components are the operational element of an application. A component can have several instances. If the correspondent node does not exist, it must be created (*CL_Create_Node* schema) before the creation of an instance. The *CL_Create_Instance* schema is shown in Figura 3.

Each port is associated to a connector which matches the desired behaviour of the port. This association can be removed if it is necessary to change the behaviour of a port or to associate this port to another connector.

Two ports (or families of ports) are linked to establish communication between the instances of components. The **link** statement is represented by the *CL_Link* schema. When two ports are connected, the connector responsible for setting up the connection stores data about the ports it is connecting. The constraints in the schema guarantee that the ports have suitable type, mode, and direction to be linked.

CL_Create_Instance

$\Delta CL\_InstanceLibrary$
ΔCL_Table
$node? : Nodes$
$component? : Component$
$machine? : Attributes$
$r! : Response$

$node? \in \text{dom } node\_parent$
$component? \in InContext$
$node? \notin InstNodes$
$InstNodes' = InstNodes \cup \{node?\}$
$node\_attr(node?)(location) = machine?$
$r! = success$

Figure 3: **CL_Create_Instance**

When an instance is activated, the configuration table is updated to keep the information that the instance has a different status. In Figure 4, we show the *CL_Activate* schema.

The reconfiguration operations are able to modify the configuration structured. For example, to interrupt the execution of an active instance (*CL_Deactivate*); to disconnect two ports, the connector responsible for the link has to eliminate the connection from the set of ports it connects (*CL_Unlink*); to delete an instance, it can not be activated (*CL_Delete*), and finally; when a component is no more necessary to the application, it can be removed from the context. In this case, it cannot have any active instance. The *CL_Remove* schema is shown in Figure 5.

CL_Activate

ΔCL_Table
$node? : Nodes$
$resp! : Response$

$node? \in InstNodes$
$(node? \notin ActiveInst) \Rightarrow ActiveInst' = ActiveInst \cup \{node?\}$
$resp! = success$

Figure 4: **CL_Activate**

CL_Remove_Component

$\Xi CL\_InstanceLibrary$
ΔCL_Table
$connector : CL\_Connector$
$c? : Component$
$resp! : Response$

$c? \in InContext$
$\forall n : Nodes \mid n \in childrens(c?) \bullet$
$(n \in ActiveInst) \Rightarrow ActiveInst' = ActiveInst \setminus \{n\} \wedge$
$(n \in InstNodes) \Rightarrow InstNodes' = InstNodes \setminus \{n\}$
$\forall node1, node2 : Nodes;\ port1, port2 : Ports \mid$
$node\_parent(node1) = c? \vee node\_parent(node2) = c? \bullet$
$\{((node1, port1), (node2, port2))\} \subseteq connector.linked\_ports \Rightarrow$
$connector.linked\_ports = connector.linked\_ports \setminus \{((node1, port1), (node2, port2))\}$
$InContext' = InContext \setminus \{c?\}$
$resp! = success$

Figure 5: **CL_Remove**

5 Case Study

The **Message Router** system presented in this section consists of a communication network connecting N senders to M receivers via a message router. Each sender is connected to one of the input ports of the router, whereas each receiver is connected to one output port [7]. The architecture of the system and the CL code for it can be seen in Figure 6. For simplification, we define only one instance of each module in the CL configuration seen in Figure 6 (b).

1. **Creating a Configuration Specification**

   The *CL_Create_System* operation is used to create the system and initialize its sets.

   The *CL_Create_Connector* specifies the connectors needed to establish communication between the instances' ports and *CL_Assign_Connector* is used to associate instances of ports to connectors. Having specified the components, interfaces and assignment, it is possible specify the operations that build the configuration program.

   In this example, the *CL_Define_Context* operation is invoked three times and the input variables have the values: router, sender and receiver.

   Then, the *CL_Create_Instance* operation is used to create each instance. In the example, the input variables have the values: {rt, router}, {sd, sender} and {rcv, receiver}. We assume that all the nodes are executed in the same machine.

   Each link command in the configuration program corresponds to a *CL_Link* operation. The input variables of that schemas are the names of the nodes and the ports

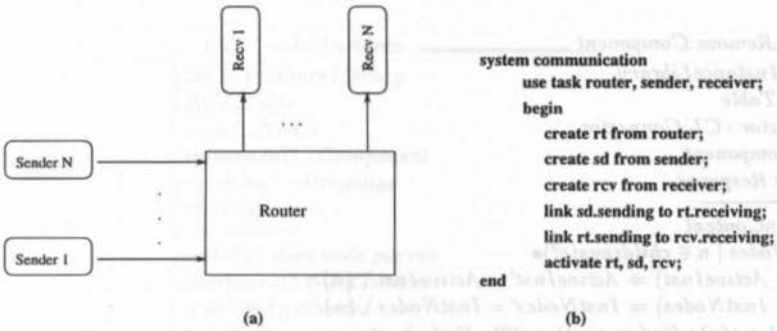

Figure 6: The Message Router System

to be linked, which in the example correspond to: {sd, rt, sd.sending, rt.receiving} and {rt, rcv, rt.sending, rcv.receiving}.

Finally, the *CL_Activate* operation is used to activate each instance: *rt*, *sd* and *rcv*.

2. **Analysing a Configuration Specification**

Our present framework enables us to perform three main types of analysis of evolving software architecture: reconfiguration operations, planned reconfiguration using reconfiguration expressions and configuration invariants. In the following, we illustrate how we applied the three types of analysis to the *Message Router* system.

(a) **Analysing Reconfiguration Operations**

Suppose that for some reason the *sd* instance has to be replaced. We have to delete *sd* and include a new instance of the *sender* component. To reconfigure the system, we have to deactivate the instance, disconnect its ports and delete it. The schemas for each operation guarantees that the restrictions will be applied. In this way, the *sd* instance will just be deactivated if it is activated and the will be delete after being deactivated and having its ports disconnected.

(b) **Analysing Planned Reconfiguration using Reconfiguration Expressions**

In the *Message Router*, the sender and the receiver should be always monitored by a router. Suppose, however, that the *rt* instance is deactivated because of a failure. To guarantee that the above constraint holds without rejecting the configuration, another instance of router should be automatically created. In our framework, we can create a schema that checks the state of the configuration when the *sd* and *rcv* instances are active but the *rt* instance is not. In that case, the configuration should automatically create another instance of the router component, link the ports to the new instance and activate the new instance. The first definition presented below says that the data requested by *CL_Create_Instance* will be provided by *Auto_Create*. The operation will

not require any input. The second definition automatically creates another instance of the router component when the condition is satisfied.

$Auto\_CL\_Create\_Instance \widehat{=} Auto\_Create \gg CL\_Create\_Instance \ {}^{0}_{9} \ CL\_Link \ {}^{0}_{9} \_CL\_Activate$

$\neg(\exists n : Nodes \mid (task^{\sim}(node\_parent(n))).id\_component = router) \Rightarrow Auto\_CL\_Create\_Instance$

(c) **Analysing Configuration Invariants**

Consistency is one of the goals of the configuration program. Many of necessary constraints are included in the configuration schema operations, but some application specific can be needed. These constraints are usually defined as configuration invariants. In the *Message Router* system, the *sd* and the *rcv* instances must be activated only if there is an instance of router already activated. In other words, the system is only valid if there exists one active instance of router as defined below:

$\exists n : Nodes \mid (task^{\sim}(node\_parent(n))).id\_component = router \wedge (n \in InstNodes \wedge n \in ActiveInst)$

6 Current and Future Work

The next step of our work is to propose and formalize a run-time reconfiguration model for dynamic reconfiguration. Our goal is to formalise aspects related to the execution of the application, verifying its state, to ensure that a reconfiguration can be done without invalidating invariant requirements of the application.

In our model, we use the idea of blocking just ports and not instances during a reconfiguration. The execution of the instance can continue normally, unless a send or receive command on the blocked port has to be executed. In this case, the execution of the instance must wait until the port is unblocked.

Each application is a top configuration and is executed by a management system represented in Figure 7.

The main manager is the executor of an application and must create the configuration and dependency table, which is used to order the reconfiguration commands. Local managers are created in each machine where it is a component in execution and they communicate to compenents and connectors to order reconfigurations.

7 Conclusions

Most of the existing languages do not support specification of dynamic architectures. Darwin is one of the exceptions. Although the work presented in [15] 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 flatten configuration, it preserves the correctness of the modules and their interconnections.

The logical framework presented by [2] allows the specification of evolving systems where the changes are static and represented by modified versions of the system. This involves encoding in the logical formalism the conditions for valid software configuration

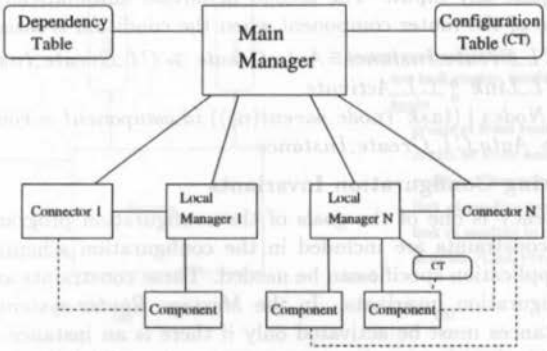

Figure 7: ZCL Management System

alterations and guarantees that the configurations remain valid after the software system is altered. Each modification generates a new version of the system. In this sense, the framework is mainly used for version control of critical systems. Also, there is not a notion of a configuration language as (re)configurations are specified by a set of axioms.

Our work presents a formal configuration language for specifying evolving distributed systems, allowing the designer to follow software development from specification to run-time, using the run-time reconfiguration model. In the current version, our framework does not include any operational aspects as to how the reconfiguration takes place but it still provides a powerful method to verify properties of the configuration. We are working on the formalization of the reconfiguration model in order to enrich the framework to prove properties of an application.

References

[1] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. *ACM Software Engineering Notes*, 18(5), December 1993.

[2] Paulo S. C. Alencar and Carlos J. P. de Lucena. A Logical Framework for Evolving Software Systems. *Formal Aspects of Computing*, 8:3–46, 1996.

[3] Robert Allen, Remi Douence, and David Garlan. Specifying Dynamism in Software Architectures. *Proceedings of Foundations of Component-Based Systems Workshop*, September 1997.

[4] Robert Allen, Remi Douence, and David Garlan. Specifying and Analysing Dynamic Software Architectures. *Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal*, March 1998.

[5] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.

[6] M. R. Barbacci, M. J. Gardner C. B. Weinstock, D. L. Doubleday, and R. Lichota. Durra: a Structure Description Language for Developing Distributed Applications. *Software Engineering Journal*, 8(2):83–94, March 1993.

[7] Paolo Ciancarini and Cecilia Mascolo. Analyzing and Refining an Architectural Style. *10th International Conference of Z Users (ZUM97), Reading, UK. April 1997. Lecture Notes in Computer Science n. 1212*, pages 349–368, April 1997.

[8] David Garlan. What is Style? *Proceedings of Dagshtul Workshop on Software Architecture*, February 1995.

[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. Ribeiro Justo and P. R. Freire Cunha. Framework for Developing Extensible and Reusable Parallel and Distributed Applications. *IEEE Conf. on Algorithms and Architectures for Parallel Processing*, pages 29–36, 1996.

[11] G. R. Ribeiro Justo, Virgínia C. de Paula, and P. R. F. Cunha. Formal Specification of Evolving Distributed Software Architectures. *International Workshop on Coordination Technologies for Information Systems (CTIS'98), in conjunction to Ninth Workshop of Database and Expert Systems Applications (DEXA'98)*, August 1998.

[12] G.R.R. Justo, P.R.F. Cunha, and V.C.C. de Paula. Distributed Systems Programming Based on Configuration-Oriented Languages. *XIX Informatics Latin-American Conference, Buenos Aires, Argentina*, 1993.

[13] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 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] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. *Fifth European Software Engineering Conference (ESEC'95)*, September 1995.

[16] Nenad 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.

[17] Neno 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.

[18] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[19] Mark Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0 - A Language for Specifying Software Architecture Hierarchies. Technical report, Computer Science Laboratory SRI International, SRI-CSL-97-01, March 1997.

[20] Virgínia C. de Paula, G. R. Ribeiro Justo, and P. R. F. Cunha. ZCL: A Formal Framework for Specifying Dynamic Distributed Software Architectures. *Nineth European Workshop on Dependable Computing (EWDC9). Gdansk, Poland*, May 1998.

[21] Virgínia Carvalho Carneiro de Paula. ZCL: A Formal Framework for Specifying Dynamic Distributed Software Architectures. Technical report, Department of Informatics, Federal University of Pernambuco, Recife, Brazil. PhD Thesis Proposal., July 1998.

[22] Peyman 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.

[23] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. *Proceedings of the International Conference on Software Engineering (ICSE'98), Kyoto, Japan*, April 19-25 1998.

[24] Matthias Radestock and Susan Eisenbach. Formalising System Structure. *International Workshop on Software Specification and Design, IEEE Computer Society Press*, pages 95–104, 1996.

[25] M. D. Rice and S. B. Seidman. A Formal Model for Module Interconnection Languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, January 1994.

[26] Michael D. Rice and Stephen B. Seidman. Using Z as a Substrate for an Architectural Style Description Language. Technical report, Department of Computer Science, Colorado State University, USA. CS-96-120, September 1996.

[27] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. *Spring-Verlag Lecture Notes in Computer Science*, 1000, 1995.

[28] Mary Shaw and David 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.

[29] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[30] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall, 1989.