Incorporating Non-functional Requirements into Software Architectures

Nelson S. Rosa[1][2][*], George R. R. Justo[1], and Paulo R. F. Cunha[2]

[1] University of Westminster, Centre for Parallel Computing,
115 New Cavendish Street, London W1M 8JS, UK
[2] Universidade Federal de Pernambuco, Centro de Informática,
Av. Prof. Luiz Freire, s/n - Cidade Universitária
CEP 50732-970 Recife, Pernambuco, Brasil

Abstract. The concept of software architecture has created a new scenario for incorporating non-functional and transactional requirements into the software design. Transactional and non-functional requirements can be included in an architecture-based software development through formal approaches in which first-order and temporal logic are utilised to deal with them. In this paper, we present an approach in which transactional and non-functional requirements are formally incorporated into a special class of software architectures, known as dynamic software architectures. In order to demonstrate how this proposal can be utilised in a real application, an appointment system is presented.

1 Introduction

Functional requirements define what a software is expected to do. Non-functional requirements (NFRs) specify global constraints that must be satisfied by the software. These constraints, also known as software global attributes, typically include performance, fault-tolerance, availability, security and so on. Closely related to NFRs[1], transactional requirements state the demand for a consistent, transparent and individual execution of transactions by the system. The well known ACID properties, Atomicity, Consistency, Isolation and Durability, summarise these transactional requirements.

During the software development process, functional requirements are usually incorporated into the software artifacts step by step. At the end of the process, all functional requirements must have been implemented in such way that the software satisfies the requirements defined at the early stages. NFRs, however, are not implemented in the same way as functional ones. To be more realistic, NFRs are hardly considered when a software is built. There are some reasons that can help to understand why it is too difficult to consider these requirements into

---

[1] Throughout the paper the term NFRs is used to refer to both non-functional and transactional requirements.

the software development: firstly, NFRs are more complex to deal with; secondly, they are usually very abstract and stated only informally, rarely supported by tools, methodologies or languages; thirdly, it is not trivial to verify whether a specific NFR is satisfied by the final product or not; fourthly, very often NFRs conflict with each other, e.g. availability and performance; and finally, NFRs commonly concern environment builders instead of application programmers.

In spite of the difficulties mentioned above, the concept of software architecture and new developments in component-based technologies offers a new perspective as NFRs can be incorporated in software development. The software architecture principles facilitate the incorporation and analysis of transactional requirements [9] and certain NFRs such as security [5], fault-tolerance [6] and multiply NFRs [1] at early stage of the development. Component-based technologies like EJB (Enterprise JavaBeans [8]) support NFRs and transactional requirements adopting the strategy of separation of concerns, in which functional and NFRs are clearly separated in the implementation.

This paper presents a formal model for specifying NFRs and show how this model has been incorporated into an existing formal framework [3] for specifying dynamic software architectures. Dynamic software architectures are usually applied to dynamic reconfigurable distributed systems, the main feature of which is the possibility of changing the system structure (configuration) during its execution. Models for describing this kind of system contain abstractions and mechanisms that allow dynamic configuration to be explicitly stated in the model. These models also form a basis for describing dynamic software architectures [3].

This paper is organised as following: Section 2 describes the configuration model CL and its formal framework ZCL. Section 3 presents the formal model for describing non-functional and transactional requirements and describes its incorporation into the ZCL framework. In order to illustrate the use of the formal model, a case study is shown in Section 4. Finally, the last section presents the conclusions and some directions for future work.

2 ZCL Framework

This section presents the CL model and the ZCL framework, which are the basis for describing dynamic software architectures, that we will adopt.

2.1 CL Model

CL is a configuration model for describing dynamic configuration systems, which are systems that require its structure (or configuration) to change during execution without having to interrupt the entire system [2]. In order to model this kind of systems, a set of abstractions is defined in the CL model: modules, ports, instances, connections and configurations. Modules are software components and may be classified into primitive (task) and composite (group). Each component has an interface, which is a set of ports (entry and exit ports), that allows the

component to communicate with its external environment. Components can be interconnected each other through its ports, defining the (system) configuration.

Using the CL model, a dynamic software architecture (or configuration) is built in five main steps: storing the components in the component library; selecting components from a component library; creating instances of these components; linking the ports of the instances; and finally activating the instances.

2.2 ZCL Framework

The ZCL framework is a formal framework based on the CL model, which formally incorporates the elements of the model. The framework consists of the CL abstractions, configuration commands and the execution model specified in the Z language [7]. Observe that in this paper we only consider structural aspects of the ZCL framework. It means that the execution model is not covered in our formal model.
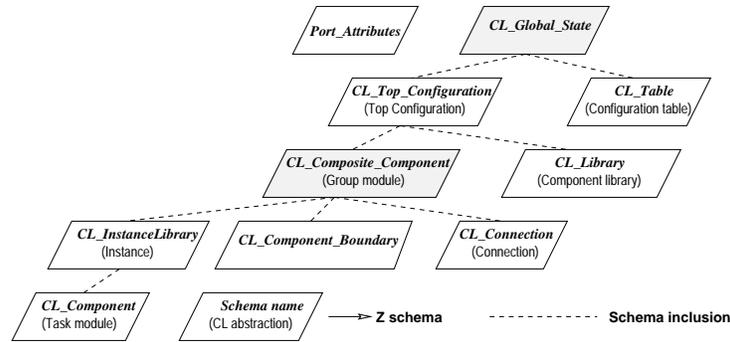

Fig. 1. Overview of the ZCL framework schemas

Figure 1 shows an overview of the Z schemas defined for modelling the main concepts of the ZCL model. The *CL_Component* schema is included in the definition of the *CL_InstanceLibrary* schema. The *CL_InstanceLibrary* schema together with *CL_Connection* schema and *CL_Composite_Boundary* schema make up the *CL_Composite_Component* (group module) schema.

$$
\begin{array}{l}
\_\,CL\_Component _____ \\
component\_attr : Indices \nrightarrow Attributes \\
interfaces : \mathbb{F}\,PortNames \\
port\_attr : PortNames \nrightarrow Port\_Attributes \\
_____ \\
\mathrm{dom}\ port\_attr \subseteq interfaces
\end{array}
$$

The *CL_Composite_Component* schema is defined as a state schema. It keeps information about its task components and group components, links between

them (connections) and a set of virtual ports[2] that belongs to its interface. The *Composite_Component* and the *CL_Library* (component library) schemas, in which predefined components are stored, form the top configuration (see Fig. 1). Finally, the *CL_Top_Configuration* schema and the *CL_Table* schema are used for storing information about the context, instances and their status and form the global state of a configuration defined by the *CL_Global_State* schema.

$$
\begin{array}{|l}
\_\,CL\_Global\_State \,\underline{\hspace{4cm}} \\
CL\_Top\_Configuration \\
CL\_Table \\
\hline
(InContext \subseteq \mathrm{dom}\; tasks \lor InContext \subseteq \mathrm{dom}\; groups) \\
InstNodes \subseteq \mathrm{dom}\; node\_parent
\end{array}
$$

In addition to the state schemas described above, the ZCL framework also defines operations to be applied to these schemas. A more detailed description of the ZCL framework, including the execution model, can be found in [3].

3 Formalising and Incorporating NFRs into Dynamic Software Architectures

In this section, we present the formal model for describing transactional and non-functional requirements and its incorporation into the ZCL framework.

3.1 Formalising NFRs

In order to formalise NFRs, we follow an approach where we first select which NFRs are considered in the formalisation, define a strategy for specifying the NFRs considering that the formal notation to be utilised is Z, and finally specify the NFRs.

Selecting NFRs. The first step to be carried out in the formalisation of NFRs is to decide which ones will be considered. It is a consequence of a very distinctive nature of NFRs, in which a wide variety of aspects such as modifiability and fault-tolerance are categorised as non-functional properties. The IEEE/ANSI 830-1993, IEEE Recommended Practice for Software Requirements Specifications, defines thirteen non-functional requirements that must be included in the software requirements document: performance, interface, operational, resource, verification, acceptance, documentation, security, portability, quality, reliability, maintainability and safety. Kotonya [4] classifies these requirements into three main categories: Product requirements, Process requirements and External requirements. Product requirements specify the desired characteristics that a system or subsystem must possess. Process requirements put constraints on the development process of the system. External requirements are constraints

---

[2] Virtual ports define the interface of a composite module.

applied to both the product and the process and which are derived from the environment where the system is developed.

Adopting this classification, we define the following criteria for selecting NFRs:

- The first criteria adopted is to consider only the first category of NFRs, namely Product requirements. This decision is based on the fact that we are interested in NFRs related to runtime properties, because the nature of dynamic systems demands special considerations during the system execution. For instance, if a component must be replaced dynamically, it is necessary to know if the new component satisfies the same requirements of performance, availability, fault-tolerance and so on of the original one.
- The second important decision relates to Product requirements. Our decision is based on the principle of how precisely these requirements could be described. Essentially, requirements were chosen according to the possibility of formulating them precisely and thus quantify them.
- The third criteria is directly related to the increasing support provided by implementation environments, e.g. Enterprise JavaBeans [8], for implementing NFRs. NFRs potentially provided by implementation environments are strong candidates because their implementation is more concrete.

Following above three criteria, three Product requirements were chosen, namely performance, reliability, security, in addition to the transactional requirements. For each selected requirement, we also identified its key attributes. Two attributes related to performance were defined: processed transaction per second; and the response time that a system should respond a user request. In terms of reliability, we define the rate of occurrence of failure and mean time to failure as the key attributes. For safety, user identification has been identified as the key attribute to protect the system against unauthorised access. Finally, the transactional requirements are represented by the ACID properties: Atomicity, Consistence, Isolation and Durability.

Specification Strategy. After the definition of which NFRs are considered in the context of dynamic systems, it is necessary to define how these requirements must be specified using the Z language. It raises two main questions: firstly, the Z notation is based on set theory, i.e. the NFRs must be specified with the strong notion of types and sets of Z. Secondly, the specification of the NFRs must be as generic as possible to allow its assignment to different elements of the CL model.

The first point focuses on the fact that the specification of performance, availability and safety must be expressed in terms of sets, operations and constraints on these sets. Essentially, first-order logic expressions combined with the definition of sets are sufficient for modelling NFRs, instead of temporal logic [9] or first-order predicates.

Secondly, the proposed specification of NFRs is defined as Z type schemas. In this way, NFRs may be assigned to different elements of the CL model, i.e. it is possible to assign NFRs to different abstractions of the CL model, whatever its granularity. For instance, a single NFRs may be assigned to a port,

a simple component or a composite component. Additionally, it is possible to refine the NFRs specification defining additional constraints on the original type.

NFR's Specification. Following the criteria defined previously, a Z specification has been defined for three NFRs, namely performance, reliability, security and the transactional requirements represented by the ACID properties. The general overview of these Z schemas is shown in Fig. 2.
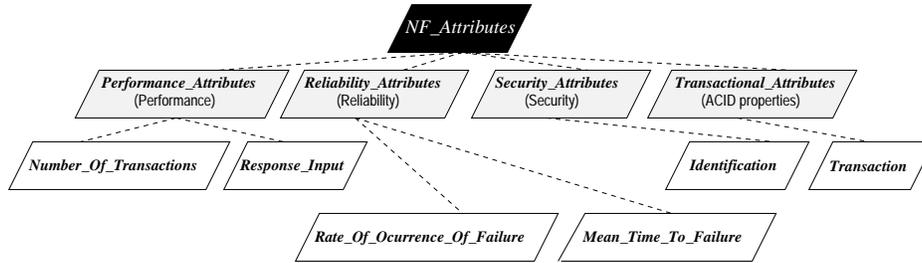

Fig. 2. Overview of Z schemas of NFRs

For each NFRs and transactional requirements a type schema was defined. For lack of space, we illustrate only one of NFRs, namely the performance. Performance is characterised by two attributes specified by the schemas *Number_Of_Transactions* and *Response_Input*, respectively. The number of transactions is specified in the *Number_Of_Transactions* schema using four variables. The variable *number_unt* defines the type of metric unit and the variable *number_value* specifies the number of processed transactions. Additionally, *number_min* and *number_max* variables define a range for the *number_value*. The predicate specifies that *number_value* should be in the range defined by *num_min* and *num_max*.

$$
\begin{array}{l}
\underline{\quad Number\_Of\_Transactions \quad\rule{4cm}{0.4pt}} \\
number\_unt : Transaction\_Unit \\
number\_value : \mathbb{N}_1 \\
number\_min : \mathbb{N}_1 \\
number\_max : \mathbb{N}_1 \\
\rule{6cm}{0.4pt} \\
number\_min \leq number\_value \leq number\_max \\
\end{array}
$$

In a similar way, the attribute response time is defined by the *Response_Input* schema.

```
┌─ Response_Input ─────────────────────
│ response_unt : Time_Unit
│ response_value : $\mathbb{N}_1$
│ response_min : $\mathbb{N}_1$
│ response_max : $\mathbb{N}_1$
├──────────────────────────────────────
│ response_min ≤ response_value ≤ response_max
```

The NFR performance is fully specified by combining its attributes in a single schema type *Performance_Attributes*.

```
┌─ Performance_Attributes ─────────────
│ attr1 : Number_Of_Transactions
│ attr2 : Response_Input
```

The schemas *Reliability_Attributes* and *Security_Attributes* are specified in a similar manner, while transactional attributes are characterised by two values defining whether the attribute is required (*ON*) or not (*OFF*). The schemas specifying each NFR are combined in the *NF_Attributes* schema type.

```
┌─ NF_Attributes ──────────────────────
│ performance : Performance_Attributes
│ reliability : Reliability_Attributes
│ security : Security_Attributes
│ transaction : Transactional_Attributes
```

3.2   Integrating NFRs into the ZCL Framework

The description of dynamic software architectures demands models that address specific questions related to dynamic configuration. The CL model has been widely utilised for this purpose. Additionally, the formalisation of this model as presented in the ZCL framework enables the formal specification and verification of dynamic reconfigurable systems. Using the ZCL framework as a tool to formally describe dynamic software architectures, we present a solution for integrating the formal specification of the NFRs into the ZCL framework: the NfZCL framework. The NfZCL framework is presented in two parts: firstly, the NFRs are integrated to the general structure of the ZCL framework; secondly, the configuration operations are extended to deal with NFRs integrated to the architecture.

Non-functional Architecture. The definition of NfZCL consists of defining how the NFRs specified in Sect. 3.1 can be assigned to the elements of the CL model. Basically, it allows the definition of dynamic software architectures with non-functional attributes. We call this kind of architecture a non-functional architecture. The non-functional architecture is specified extending the ZCL framework by assigning NFRs to each port. Observe that the decision of assigning

non-functional attributes to port, rather than component, offers more flexibility to component development. The flexibility comes from the fact that the global non-functional attributes of a component may be defined as the conjunction of the specification of its ports. If non-functional attributes were assigned to the entire component, it would not be possible to make any assumption about the performance of individual ports.

The *NF_Architecture* state schema defines a non-functional architecture as an extension to the description of dynamic architecture defined by the ZCL framework, namely *CL_Global_State* (see Fig. 1). The extension basically defines a function relating a component port with its non-functional attributes.

```
┌─ NF_Architecture ────────────────────────
│ CL_Global_State
│ nf_port : (ID_Component × PortNames) ⇸ NF_Attributes
├──────────────────────────────────────────
│ ran ( dom nf_port) ⊆ interfaces
│ dom ( dom nf_port) ⊆ dom tasks
└──────────────────────────────────────────
```

The partial function *nf_port* is the key element that assigns each port belonging to each component to the NFRs. In practical terms, this function creates a library containing ports and NFRs assigned to them. The schema asserts that each port belonging to *interface* of the component has assigned *NF_Attributes*. It is also asserted that the component must belong to the set of *tasks* of the dynamic architecture.

Non-functional Operations. As NFRs were assigned to ports, the configuration operations must be modified. For example, the operation used to link two ports needs to be extended to deal with NFRs. This is necessary because ports with NFRs incompatible can not be linked. To carried out this check, we defined one checking function for each NFR (*check_performance*, *check_reliability*, *check_security* and *check_transactional*). Each function checks whether non-functional values of *port1?* and *port2?* of distinct components *node1?* and *node2?* are compatible or not.

```
┌─ NF_Check ──────────────────────────────
│ Ξ NF_Architecture
│ node1? : Nodes
│ node2? : Nodes
│ port1? : PortNames
│ port2? : PortNames
├──────────────────────────────────────────
│ (node_parent(node1?), port1?) ∈ dom nf_port
│ (node_parent(node2?), port2?) ∈ dom nf_port
│ (let elem1 == nf_port(node_parent(node1?), port1?)
│ •let elem2 == nf_port(node_parent(node2?), port2?)
│ •((check_performance(elem1.performance_attr, elem2.performance_attr) = ok)
│ ∧(check_reliability(elem1.reliability_attr, elem2.reliability_attr) = ok)
│ ∧(check_security(elem1.security_attr, elem2.security_attr) = ok)
│ ∧(check_transactional(elem1.transactional_attr, elem2.transactional_attr) = ok)))
└──────────────────────────────────────────
```

4 Case Study: an Appointment System

In order to illustrate our approach, we specify in this section an appointment system. The appointment system follows the client-server style and realises a simple distributed appointment scheduling system. The server component provides operations that can be used to add and remove an appointment, and also an operation that returns the current time schedule. The non-functional properties of the appointment system require the set of server operations to be atomic and the isolated execution of clients that manipulate shared time schedules. In addition, it is important to ensure that changes made to the time schedules will survive subsequent system failures.

Software Architecture. The system architecture is composed by the components *AppointmentServer*, *AppointmentClient* and *DataBase*. The *AppointmentClient* component makes requests to the *AppointmenServer*, the *AppointmentServer* models the server and the *DataBase* represents the data base in which the information about appointments are stored.

Z specification. As mentioned in Sect. 2.1, five steps are usually necessary to build a configuration. The first step consists of initialising the state schemas and composing, using the Z schema composition operator $\frac{9}{9}$, with the creation of instances. The ZCL operation *CL_Create_Component* schema creates the *AppointmentServer*, *AppointmentClient* and *DataBase* components and also stores them in the library for future instantiations. After the components have been stored in the library, it is necessary to assign the non-functional properties for their ports. For example, the NfZCL operation *Associate_NF_Attributes* schema allows us to assign performance attributes (number of transactions) to the *provide* port of the *AppointmentServer*.

$assigning \widehat{=} Associate\_NF\_Attributes[c? := AppointmentServer, port? := provide,$
$number\_unt? := transactions\_per\_milliseconds, number\_value? := 8,$
$number\_min? := 1, number\_max? := 10]$

The second step consists of defining the context (defining the component types that can be used by the architecture) of the dynamic software architecture using the ZCL operation *CL_Define_Context* schema. After the definition of the context, in the third step, it is possible to create instances of components that will form the architecture. The ZCL operation *CL_Create_Instance* schema instantiates a component and places it to execute on a particular machine. Instances can be linked to define the structure of the architecture (fourth step). In this case, the ZCL operation *CL_NF_Link* schema is defined as a composition ($\frac{9}{9}$) of the NfZCL operation *NF_Check* schema and the ZCL operation *CL_Link* schema. Finally, the instances can be activated using the ZCL operation *CL_Activate* schema.

5 Conclusion and Future Works

This paper has illustrated how NFRs can be formally specified and incorporated into dynamic software architectures. Using the ZCL framework as a basis, two main contributions have been proposed: the formal specification of some NFRs and their incorporation into software architectures. From the definition of some criteria, three NFRs and transactional properties have been chosen, namely safety, availability and performance. These NFRs were formally defined in Z and incorporated into ZCL framework according to a strategy that assigns non-functional attributes to ports. Additionally, configuration operations have been extended to enable the ZCL framework to deal with NFRs.

The introduction of NFRs into the software architecture is an important step in the software design. In addition, the use of a formal model enables us to verify properties of the software in the early stages of the software design. In terms of dynamic systems, it assumes a very important role because at configuration time it is possible to check when two components have compatible non-functional properties.

As we said in the beginning of the paper, NFRs are usually considered only during the implementation. This paper has show how NFRs can be successfully incorporated at early stages of the design. It is very important, however, to relate the NFRs defined during the design to those used by the implementation environment. In this direction, our future work will focus on a refinement calculus (set of formally defined rules) for the transformation of NFRs-based software architectures.

References

[1] Issarny, V., Bidan, C., Saridakis, T.: Achieving Middleware Customization in a Configuration-based Development Environment: Experience with the Aster Prototype. Fourth International Conference on Configurable Distributed Systems, Annapolis Maryland USA (1998) 207-214

[2] Justo, G. R. R., Cunha, P. R. F.: An Architectural Application Framework for Evolving Distributed Systems. Journal of Systems Architecture **45** (1999) 1375-1384

[3] Justo, G. R. R., Paula, V. C. C. de, Cunha, P. R. F.: Formal Specification of Evolving Distributed Software Architectures. International Workshop on Coordination Technologies for Information Systems(CTIS'98), Viena Austria (1998) 548-553

[4] Kotonya, G. and Sommerville, I. Requirements Engineering: Process and Techniques. John Wiley & Sons, Inc **8** (1998) 190-213

[5] Moriconi, M., Qian, X., Riemenschneider, R. A., Gong, L.: Secure Software Architectures. IEEE Symposium on Security and Privacy. Oakland CA (1997)

[6] Saridakis, T., Issarny, V.: Fault Tolerant Software Architectures. INRIA **3350** (1998)

[7] Spivey, J.M.: Understanding Z: a Specification Language and Its Formal Semantics. Cambridge University Press (1988)

[8] Matena, V., Hapner, M.: Enterprise JavaBeans^{TM}. Sun Microsystems (1997)

[9] Zarras, A., Issarny, V.: A Framework for Systematic Synthesis of Transactional Middleware. Middleware'98. The Lake District England (1998) 257-272