Treating Non-Functional Properties of Dynamic Distributed Software Architectures

Nelson S. Rosa, Paulo R. F. Cunha
Centro de Informática
Universidade Federal de Pernambuco
50732-970 Recife, Pernambuco - Brazil
Phone: +55 81 3271 8430 Fax: +55 81 3271 8438
E-mail: {nsr,prfc}@cin.ufpe.br

George R. R. Justo
Centre for Parallel Computing
University of Westminster
115 New Cavendish Street London - W1M 8JS
Phone: +44 207 911 5000 Fax: +44 207 911 5143
E-mail: justog@cpc.wmin.ac.uk

Keywords: Software Architecture; Non-Functional Properties; Dynamic Distributed Systems; Runtime Change.

Abstract

One of key methods to increase availability of distributed systems is to allow dynamic changes to place at runtime without to stop the entire system. A critical task carried out during such changes consists of verifying and preserving the functional properties of the system. However, as important as to preserve the integrity of functional aspects, non-functional properties (e.g., *security* and *performance*) must also be preserved when modifications occur. We present a formal approach for specifying non-functional properties of dynamic distributed systems. This approach allows formal verification to be performed before a change is committed. The paper illustrates our approach with an example of a dynamic client-server application.

1 Introduction

It is increasing the number of systems in which the continuous availability is a key characteristic. For these systems, known as dynamic systems, the cost, risk and delay of shut-downing and restarting the entire application are so critical that every change must be carried out while the system still running [Oreizy 98a]. Typical dynamic systems include safety-intensive and mission-critical applications such air traffic control, telephone

switching and multimedia medical systems. Most recent commercial applications, like Internet search engines, also need to be available all time.

Changes in dynamic systems are mainly motivated by the necessity of removing bugs, enhancing the functionality or adapting the system to a new environment. For instance, to allow changes from a GSM mobile radio platform to the 3G UMTS. In practical terms, these changes consist of replacing, adding and removing parts of the application, or simply change the way in which the elements that compose the application are wired. Common to every kind of change is the necessity of preserving functional properties (such as deadlock, liveness, message loss, protocol conformance) in order to maintain the system integrity. Hence, before a change is committed, the integrity of the application must be verified to ensure that its functional properties are preserved.

As important as the preservation of functional aspects, the non-functionality of the system also must be kept unchanged after a runtime modification [Oreizy 98b]. Firstly, for most dynamic systems, non-functional properties play a critical role and their satisfaction is mandatory. For these systems, non-functional properties such as *good performance* and *secure* are as important for the correct functioning of the system as the functionality itself. Secondly, as a kind of requirement, it seems to be natural to verify the system integrity with respect to them. The absence of an explicit treatment of non-functional requirements has been a critical success factor of several software systems [Boehm 96]. Thirdly, non-functional properties have a global nature, in opposite to local effects of functional requirements. Hence, the modification of a single component of the system may affect the integrity of the entire application with respect to a particular non-functional aspect. For example, a *secure* component may be replaced by a *non-secure* one, without any kind of verification. In this case, the global nature of non-functional properties implies that the entire application will become *non-secure* just because a single *non-secure* piece of the system.

Configuration environments such as CL [Justo 99] and Darwin [Magee 95] and architectural infrastructure like ArchStudio [Oreizy 98a] do perform certain integrity verification. For instance, CL checks port compatibility before a connection is performed. In Darwin, the checking is performed verifying the compatibility of the interface types. ArchStudio uses architectural constraints imposed by the C2 architectural style [Oreizy 98c] for defining invariants that must be verified before a change is committed. If a change violates the invariant, it is not performed. However, none of these environments takes into account non-functional properties.

In this paper, we present an approach for considering non-functional properties when changes occur. Our approach adopts software architecture concepts for describing dynamic distributed systems, characterising the dynamic distributed software architectures [Kramer 97]. It concentrates on verifying the integrity of the application with respect to its non-functional properties, i.e., if a system is *secure* before a runtime change, it must still be *secure* after it. Essentially, these properties are defined as invariants that cannot be violated during changes. In order to perform this task, we formally describe both the non-functional properties, including their notion of compatibility and correlation, and the software architecture concepts. These formal descriptions are thereby incorporated into operations used to change the system at runtime.

This paper is organised as follows. Section 2 introduces basic concepts related to non-functional properties and software architecture. Next section, Section 3, presents the formal definition of basic concepts introduced in the previous section. Section 4 presents

the operations used to dynamically change the system considering the non-functional properties. In Section 5, we apply our approach to a dynamic client-server system. Finally, the last section presents the conclusions and some directions for future work.

2 Basic Concepts

In our approach, a dynamic distributed system is described in terms of software architecture elements such as components, connectors and configuration. To these elements, non-functional properties are assigned in order to characterise their quality attributes.

2.1 Non-functional Requirements

Functional requirements define *what* a software is expected to do, while non-functional requirements[1][2] (NFRs) specify global constraints on *how* the software operates or how the functionality is exhibited [Chung 99]. Functional requirements usually have localised effects, i.e., they affect only the part of the software addressing the functionality defined by the requirement. During the software development process, functional requirements are usually incorporated into the software artefacts 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, have a global nature, which means that to satisfy a NFR may affect several design components.

NFRs have a very distinctive nature, 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 [Kotonya 98] 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 applied to both the product and the process, which are derived from the environment where the system is developed.

In order to formalise NFRs, we adopts the set of abstractions proposed in [Rosa 00, Rosa 01], in which non-functional information is modelled by two abstractions: NF-Attribute and NF-Requirement. A NF-Attribute models both non-functional characteristic of the software that can be precisely measured out (*performance*) and non-functional features that cannot be quantified, but may be defined as present in the software in a certain level (*security*). A NF-Requirement is a constraint over a NF-Attribute, e.g., *Good Performance* is a constraint on the NF-Attribute *performance*, which defines that the *performance* must reach a certain value.

[1] Also referred to as goals [Mylopoulos 92], *ilities* [Filman 98], software quality factors [Ghose 99] or commonly quality attributes.

[2] NFRs, quality attributes and non-functional properties are terms used interchangeably throughout the paper.

2.2 Software Architecture

The software architecture [Shaw 96] is the highest abstract description of a software design, which is defined at the initial stages of the software development. Software architectures are commonly described in terms of three basic abstractions: components, connectors and configurations. Components represent a wide range of different elements, from a single client to a database, and have an interface (made up of ports) used to communicate the component with the external environment. Connectors represent communication elements between components. The configuration describes how components and connectors are wired. A traditional view of software architecture is shown in Figure 1, where three components (*CompA*, *CompB*, *CompC*) and two connectors ($C1$, $C2$) make up the configuration. Each component has ports that compose its interface: p_1 and p_2 of *C1*, p_4 of *CompB* and p_3 of *CompC*.

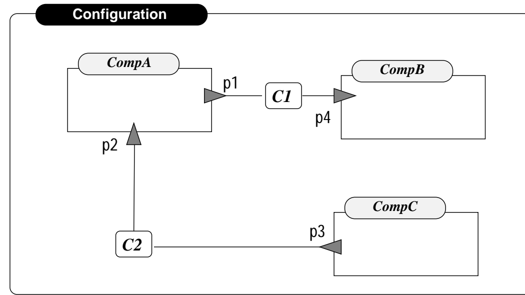

Figure 1: Software Architecture

Essentially, the software architecture presents a description of the software where "computation" or "functionality" (included in components) and "communication" (modelled by connectors) are clearly separated. In order to describe software architectures, languages specially designed for this purpose, namely Architecture Description Languages (ADLs), replace box-and-arrows diagrams and natural languages. This class of languages has some key characteristics such as components and connectors as first-class elements, ability for expressing the NFRs and the focus on the structure of software [Medvidovic 00].

3 Formalising Basic Concepts

NFRs and architectural elements presented in the previous sections are formalised through first-order logic and set theory concepts. The notation used to formalise the basic concepts consists of the logic operators $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$; $=_{df}$ means "is defined by"; $[P]$ means that P is true for all values of variabels in its alphabet; $S \setminus T$ is the set minus operator; and (x, y, z) defines a tuple.

3.1 Formalising Non-functional Requirements

Definition [**NF-Attribute**] A NF-Attribute is specified as a set of typed variables as follows

$$nfa =_{df} \{ \; var_1 : Type_1, ..., var_n : Type_n \; \}$$

For example, the NF-Attribute *performance* is traditionally defined in terms of *space* and *time* [Chung 99] as shown bellow

$$performance = \{ \ space : integer, time : real \ \}$$

Definition [NF-Requirement] A NF-requirement is a predicate[3] (P) over a NF-Attribute as shown bellow

$$nfr =_{df} \mathrm{P}(nfa)$$

For instance, a NF-Requirement over the NF-Attribute $performance$ may be defined as

$$Good_Performance \ (performance) = time < 0.045$$

A NF-Requirement may also be defined as a logical expression that includes two or more predicates over different NF-Attributes. For instance, the security of systems is usually defined having a certain level (*SECURITY_LEVEL*). Hence, the NF-Requirement *Fast_and_Secure* is defined as follows ($performance$ and *Good_Performance* as defined before)

$$security = (level : SECURITY_LEVEL)$$

$$Secure \ (security) = (\text{level} = \text{Level_4})$$
$$Fast_and_Secure \ (performance, security) = Good_Performance \ (performance) \wedge Secure(security)$$

In addition to these two basic definitions, we also consider the idea of correlation among NF-Attributes, as proposed in the NFR Framework [Chung 99]. According to this framework, NF-Attributes are usually correlated either positively or negatively. A "positive" correlation means that the NF-Attribute acts "in favour of" another one, while a "negative" correlation has an opposite effect. Essentially, the notion of correlation serves to capture the relation among NF-Attributes or how one affects others.

Definition [Correlation] A correlation is defined as an implication between constraints imposed on distinct NF-Attributes as follows

$$\text{Correlation}(nfa_1, nfa_2) =_{df} P_1(nfa_1) \Rightarrow P_2(nfa_2), \text{ where } nfa_1 \neq nfa_2$$

For example, an improvement in the *security* likely has a negative effect on the *performance*, as additional actions must be performed to improve the *security*

$$High_Security \ (security) = (\ level = 5 \ )$$
$$Low_Performance \ (time) = (\ time > 1.47 \ )$$

$$High_Security \Rightarrow Low_Performance$$

[3] A predicate is formally defined as an equation or an inequation or a collection of formulae. It may be defined as the set of all tuples that satisfy the predicate $\{(x, y, ..., z\} \mid P(x, y, ..., z)\}$.

Definition [Compatibility] Two NF-Requirements are compatible if they constrain the same NF-Attribute and the evaluation of both predicates produces the same value ([$P(x)$] means evaluation of P for every value of x)

$$nfr_1 = P_1(nfa_1) \text{ and } nfr_2 = P_2(nfa_2)$$

$$\text{Compatibility } (nfr_1, nfr_2) =_{df} (nfa_1 = nfa_2) \wedge [P_1(nfa_1)] \Leftrightarrow [P_2(nfa_2)]$$

3.2 Software Architecture

The architectural elements mentioned in Section 2.2, namely port, interface, component, connector and configuration are formally defined in the following.

Definition [Port] A port is the point in which the component communicates with the external environment. It is described through the definition of the direction of the port, namely input port or output port.

$$PORT_DIRECTION ::= input \mid output$$

$$port =_{df} (dir_type : PORT_DIRECTION)$$

It is worth noting that additional elements may be used to describe a port [Paula 98]. However, this description only including the port direction is sufficient for the scope of this paper.

Definition [Interface] The interface is defined a set of ports as follows

$$interface =_{df} \{port_1, ..., port_n\}$$

Definition [Component] The component description includes an interface and three different non-functional requirements as shown bellow

$$component =_{df} (comp_{interface}, comp_{Pnfr}, comp_{Rnfr}, comp_{Cnfr})$$

The interface expresses the functionality provided by the component; $Comp_{Pnfr}$ is the quality of the computation performed by the component; $comp_{Rnfr}$ refers to the expected quality of the computation carried out by other components connected to it; $comp_{Cnfr}$ defines the quality requirements expected from the communication (in the software architecture, the connector). For instance, a component may have a *good performance*, require a *secure* communication and demand *fast computation* from the components connected to it.

Definition [Connector] The connector has an interface and provides communication services with a certain quality, as specified bellow

$$connector =_{df} (conn_{interface}, conn_{nfr})$$

The description of a connector may also involve more elements [Allen 97] ones the defined above are sufficient for our purpose.

Definition [Configuration] The configuration ($configuration$) is defined in terms of a set of components ($components$), a set of connectors ($connectors$), connections between these elements ($connections$), a global invariant ($configuration_{nfr}$) and possible correlations ($correlations$) between the NF-Attributes constrained in the software architecture.

$$
\begin{aligned}
configuration \quad =_{df} \quad & (components,\ connectors,\ connections, \\
& configuration_{nfr}(components, connectors), \\
& correlations)
\end{aligned}
$$

For example, the software architecture shown in Figure 1 may be defined as follows

$$components = \{CompA,\ CompB, CompC\}$$

$$connectors = \{C1, C2, C3\}$$

$$connections = \{(CompA, C1, CompC), (CompA, C2, CompC)\}$$

$$
\begin{aligned}
& configuration_{nfr}(components) = \\
& \qquad \forall comp \in components \bullet \text{ Compatibility}(comp_{Pnfr}, P_1(nfa_1))
\end{aligned}
$$

$$correlations(nfa_n, nfa_m, nfa_s) = (P_n(nfa_n) \Rightarrow P_m(nfa_m)) \wedge (P_n(nfa_n) \Rightarrow P_s(nfa_s))$$

4 Formalising Runtime Change Operations

As mentioned in Section 1, operations of adding, removing, replacing and reconfiguring are used to change the dynamic distributed software architecture. These four operations are formalised considering some assumptions related to the non-functional properties:

- *Non-functional properties within connectors* : The non-functionality spreads throughout components and connectors, while the functionality itself is restrict to the component. Being a communication element, the functional integrity is not verified when a connector is replaced. However, it is not true in relation to NFRs, as they are also present in the connector and must be verified when any runtime operation involves a connector;
- *Role of connectors* : Software architecture principles define that components only communicate through a connector, never directly. Consequently, operations involving connectors are usually more complicated, as they involve NFRs of three architectural elements, i.e., NFRs of the connector itself and two components linked by the connector;
- *Two components and one connector* : We consider each connector linking only two components. This assumption is merely taken in account in order to clarify the presentation of the change operations, without lost of generality. Each operation may be easily extended for considering that a single connector binds more than two components. It is carried out checking the compatibility conditions for every (not simply two) component using the connector;

- *NFRs as invariants* : NFRs are invariants explicitly considered for preserving the system integrity. Hence, before a change is committed, these invariants are verified in order to prevent changes that violate them;
- *Software architecture versus configuration models* : Dynamic distributed systems are usually specified according to configuration models such as CL and Darwin. However, instead of use these models, we adopt a similar approach as proposed by Oreizy [Oreizy 98a]. According to this approach, software architecture elements are used to describe dynamic systems. Benefits of this approach comes with the explicit separation of computation and communication, control over changes in the hands of architects and the use of a software architecture as a representation of the system;
- *Focus on architectural level* : Our approach does not address the mechanisms provided for implementing runtime change operations, but focuses on analysing/modelling change operations at architectural level in order to ensure the application integrity considering non-functional requirements. Implementation environments such as Darwin, CL and ArchStudio are responsible for the runtime changes;
- *Functional compatibility* : The functional compatibility is tested using the interface defined for each component. In particular, we consider that interface compatibility means behavioural compatibility. It is worth noting that assumption related to functional compatibility are out of the scope of this paper; and
- *Separation of non-functional properties* : The software architecture concepts explicitly separate communication and computation concerns. Following this principle, we also divide explicitly non-functional requirements related to the computation (assigned to components) and ones present in the communication (associated to connectors).

Considering these assumptions and using the formal definition of NFRs and architectural concepts presented in sections 3.1 and 3.2, we define what have to be checked in each change. We provide seven change operations, namely **addComp, addConn, remComp, remConn, repComp, repConn** and **configuration**, and the conditions that must be satisfied before their execution. Each operation changes the configuration, while the condition defines precisely the non-functional properties that must be checked. Additionally, it is worth noting that these operations are considered to be atomic. This fact is important as the system may be in an inconsistent state when the change is being performed, unless the atomicity is guaranteed.

Adding a Component A component is added at runtime in order to enhance the system functionality to meet changing user needs. When a new component is added to the dynamic system, it is necessary to check the compatibility between NF-Requirements of the new component and the component to which it will be connected, and the compatibility with the communication requirements provided by the connector. Additionally, the NFRs of the new component are checked in order to avoid violating the predicate defined for the configuration. Figure 2 depicts the addition of the component *NewComp*. It is worth noting that a new connector is not effectively added (the set *connectors* is not altered), but simply a new instance of the connector $C2$ is created as it is always necessary a connector between every two components in the software architecture.

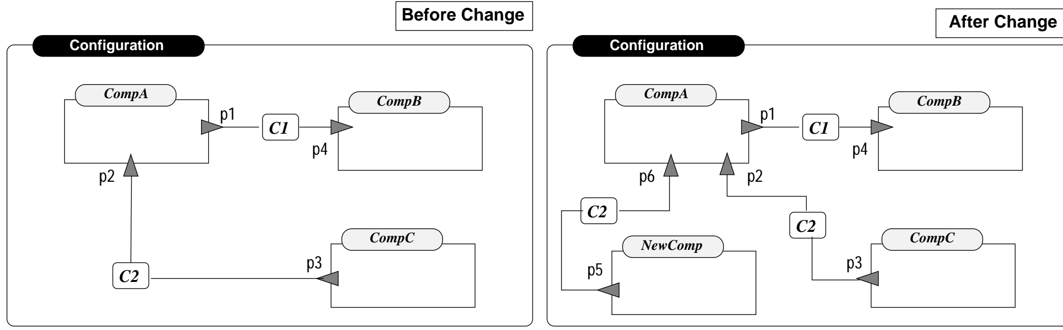

Figure 2: Adding a Component

The operation **addComp** is used to add a component. It is executed if the condition addCompCondition is satisfied. In this case, a new component is added to the set of components and a new connection is added to the set of connections. Otherwise, the configuration is not changed. This operation is defined as follows

$$\textbf{addComp}\ (configuration, NewComp) = \begin{cases} components \cup \{NewComp\} \text{ and}^5 \\ connections \cup \{(CompA, C2, NewComp)\} & \textbf{if } \mathsf{addCompCondition} \\ \\ \text{no action (configuration not changed)} & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathsf{addCompCondition} = \ & \text{Compatibility } (NewComp_{Pnfr}, CompA_{Rnfr}) \wedge \\ & \text{Compatibility } (NewComp_{Rnfr}, CompA_{Pnfr}) \wedge \\ & \text{Compatibility } (NewComp_{Cnfr}, C2_{nfr}) \wedge \\ & configuration_{nfr}\ (NewComp) \wedge correlation \end{aligned}$$

Adding a Connector New connectors are added to communicate components, which were not previously connected. A basic condition to be verified when a new connector is added refers to the communication requirements provided by the connector must be compatible with ones required by the components being connected. Figure 3 shows the addition of the connector *NewConn* connecting *CompB* and *CompC*.

The operation defined for adding a connector, namely **addConn**, is similar to one used to add a component, except for the modified set and the addConnCondition.

$$\textbf{addConn}\ (Conf, (CompA, NewConn, CompB)) = \begin{cases} connectors \cup \{NewConn\} \text{ and} \\ connections \cup \{(CompB, NewConn, CompC)\} & \textbf{if } \mathsf{addConnCondition} \\ \\ \text{no action (configuration not changed)} & \text{otherwise} \end{cases}$$

where

[5] It means that an opperation follows another one.

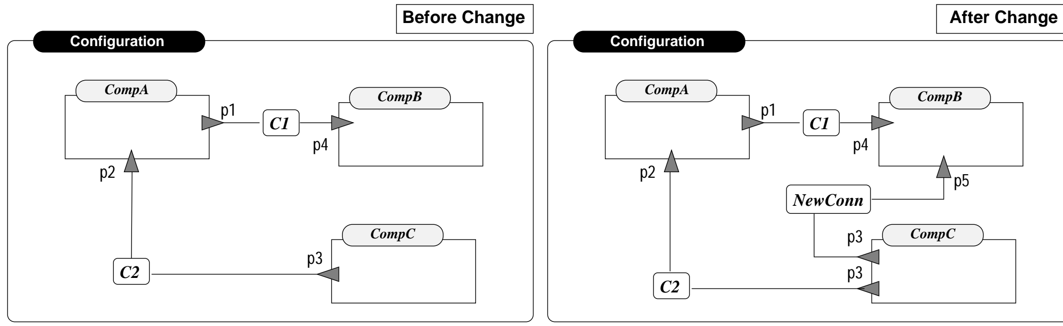

Figure 3: Adding a Connector

$$\mathsf{addConnCondition} = \text{Compatibility } (NewConn_{nfr}, CompA_{Cnfr}) \wedge \text{Compatibility } (NewComp_{nfr}, CompB_{Cnfr}) \wedge configuration_{nfr}\ (NewConn) \wedge correlation$$

Removing a Component Components are removed in order to eliminate unnecessary behaviour. Usually, the unnecessary behaviour is a consequence of recent added functionality. Issue to be considered when a component is removed refers to the fact that some components may be responsible for a specific non-functional characteristic of the software. For instance, the component may take responsibility for the encryption. In this particular case, the component cannot be removed.

A component is removed using the operation **remComp**. If the condition remCompCondition is satisfied, the component is removed (through the set difference operator "\") from the set of components that make up the configuration and the connection including $OldComp$ is removed from the set of connections (*connections*).

$$\textbf{remComp}\ (configuration, OldComp) = \begin{cases} components \setminus \{OldComp\} \text{ and} & \\ connections \setminus \{(OldComp, C2, CompA)\} & \textbf{if } \mathsf{remCompCondition} \\ & \\ \text{no action (configuration not changed)} & \text{otherwise} \end{cases}$$

where

$\mathsf{remCompCondition} = OldComp$ does not implement any task related to a NF-Attribute.

Removing a Connector Removing a connector is necessary when the connection between two components is not needed anymore. The operation **remConn** does not place any condition to be applied to remove a connector

$$\textbf{remConn}\ (configuration, OldConn) = connectors \setminus \{OldConn\} \text{ and } connections \setminus \{(CompB, OldConn, CompC)\}.$$

Replacing a Component Components are usually replaced in order to correct an undesired behaviour or to enhance a system functionality. NFRs of the new component must be compatible with the previous one. Figure 4 shows the replacement of *OldComp* by *NewComp*.

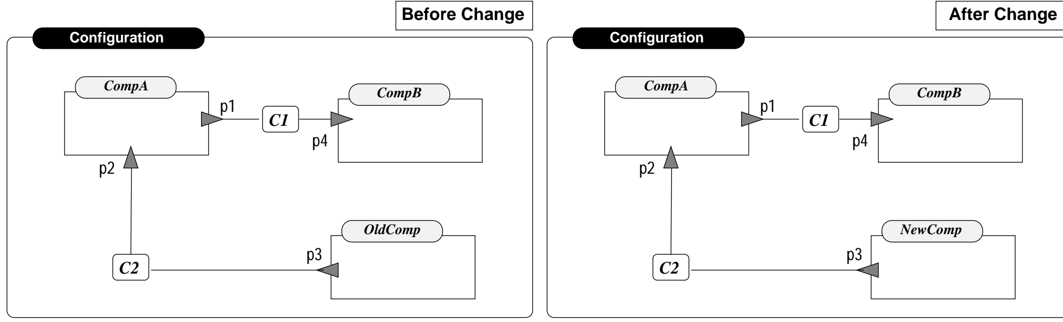

Figure 4: Replacing a Component

The replacement operation **repComp** is performed removing the old component ($OldComp$) and adding the new one. Both operations are performed if the condition repCompCondition is satisfied.

repComp ($configuration$, $OldComp$, $NewComp$) =

$$\begin{cases} configuration \setminus \{OldComp\} \text{ and} & \\ configuration \cup \{NewComp\} & \textbf{if } \text{repCompCondition} \\ & \\ \text{no action (configuration not changed)} & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \text{repCompCondition} = \; & \text{Compatibility } (OldComp_{Pnfr}, NewComp_{Pnfr}) \wedge \\ & \text{Compatibility } (OldComp_{Rnfr}, NewComp_{Rnfr}) \wedge \\ & \text{Compatibility } (OldComp_{Cnfr}, NewComp_{Cnfr}) \wedge \\ & configuration_{nfr} \; (NewComp) \end{aligned}$$

Replacing a Connector Connectors are replaced when it is needed to improve the communication between two components. As for components, the replacement operation **repConn** must verify the compatibility of NFRs of the new component and the new one, as defined in the condition repConnCondition

repConn ($configuration$, $OldConn$, $NewConn$) =

$$\begin{cases} connectors \setminus \{OldConn\} \text{ and} & \\ connectors \cup \{NewConn\} & \textbf{if } \text{repConnCondition} \\ & \\ \text{no action (configuration not changed)} & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \text{repConnCondition} = \; & \text{Compatibility } (OldConn_{nfr}, NewConn_{nfr}) \wedge \\ & configuration_{nfr} \; (NewConn) \wedge correlation \end{aligned}$$

Reconfiguration The reconfiguration means to recombine existing functionality to modify overall system behaviour. As there is a connector between every two components, the reconfiguration may be performed by changing the connector bindings, i.e., it

consist of altering the set *connections* by removing an old connection and adding a new one. The problem related to the reconfiguration is to avoid putting together components with different NFRs and to use a connector with incompatible NFRs in relation to the components. Figure 5 depicts a reconfiguration in which the connection between $CompA$ and $CompC$ is removed and a new connection is performed between $CompB$ and $CompC$.

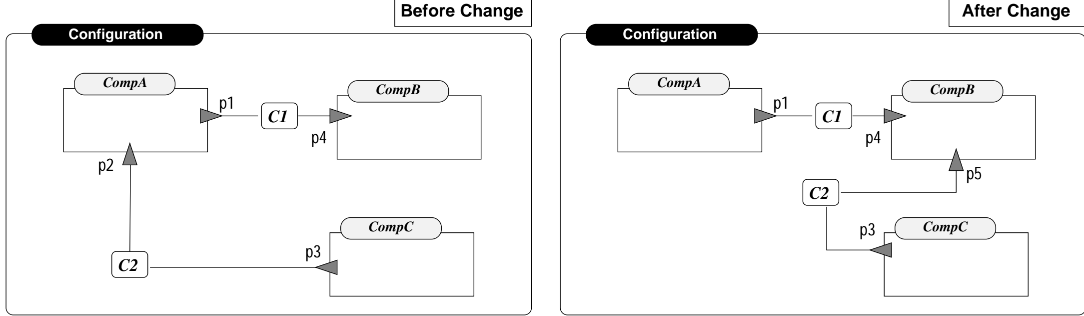

Figure 5: Reconfiguration

The reconfiguration operation, namely **reconfiguration**, consists of removing a particular connection and adding a new one, which expresses the new desired configuration. This operation is defined if recCondition is satisfied.

$$\textbf{reconfiguration}\ (Conf,\ (CompA,\ C2,\ CompC),\ (CompB,\ C2,\ CompC)) = \begin{cases} connections \setminus \{(CompA, C2, CompC\} \text{ and} \\ connections \cup \{(CompB, C2, CompC)\} & \textbf{if}\ \mathsf{recCondition} \\ \\ \text{no action (configuration not changed)} & \text{otherwise} \end{cases}$$

where

$$\mathsf{recCondition} = \begin{array}{l} \text{Compatibility}\ (CompB_{Rnfr},\ CompC_{Pnfr}) \wedge \\ \text{Compatibility}\ (CompB_{Pnfr},\ CompC_{Rnfr}) \wedge \\ \text{Compatibility}\ (CompB_{nfr},\ C2_{nfr}) \end{array}$$

5 Case Study - Dynamic Client-server

In order to illustrate our approach, we apply the formalization and checking of system integrity proposed in the previous sections to a client-server application. In fact, this application is a dynamic distributed client-server, in which the server can be dynamically changed by another one. In terms of NFRs, the communication between elements of the application must be *Secure* and the application must be *Fast*. Figure 6 depicts the application of the replacement operation to the server.

Initially, we describe the NF-Attributes defined for the application. These NF-Attributes include *security*, *accuracy* and *performance*. The *security* is defined in terms of security levels; the *accuracy* is simply defined as present or not; and the *performance* is specified in terms of the number of transactions processed per second. We define the NF-Requirements *Secure*, *Accurate* and *Fast* in terms of these NF-Attributes.

$$SECURITY_LEVEL ::= Level_1 \mid Level_2 \mid Level_3 \mid Level_4 \mid Level_5$$

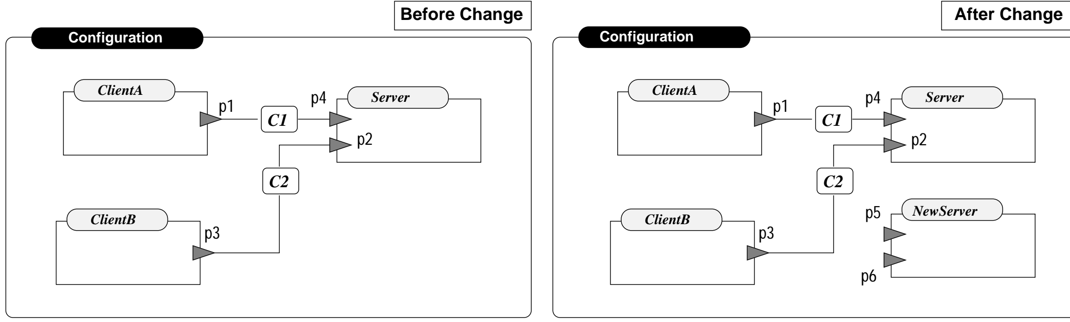

Figure 6: Dynamic Client-server Software Architecture

$security$ = (level : $SECURITY_LEVEL$)
$performance$ = (transactions_processed : $integer$)
$accuracy$ = (value : $boolean$)

$Secure$ ($security$) = (level = $Level_5$)
$Secure_Communication$ ($security$) = (level = $Level_2$)
$Fast$ ($performance$) = (transactions_processed > 20)
$Accurate$ ($value$) = $value$

The dynamic distributed software architecture contains three components and two connectors. The *ClientA* has a port p_1, its computation does not have any non-functional property associated to (*none*) and requires a *Fast* and *Accurate* server; the *ClientB* has a port p_3, does not have any non-functional property assigned to (*none*) and requires a *Secure* server. Both component require a *Secure_Communication*; the server provides a *Secure*, *Fast* and *Accurate* computation, while requires a *Secure_Communication*.

$$
\begin{array}{lll}
ClientA & = & (\{p_1\},\ none,\ Fast \wedge Accurate,\ Secure_Communication) \\
ClientB & = & (\{p_3\},\ none,\ Secure,\ Secure_Communication) \\
C1 & = & (C1_{interface},\ Secure_Communication) \\
C2 & = & (C2_{interface},\ Secure_Communication) \\
Server & = & (\{p_2, p_4\}, \\
 & & Secure \wedge Fast \wedge Accurate, none, \\
 & & Secure_Communication)
\end{array}
$$

$$
\begin{array}{lll}
configuration & = & (\{ClientA, ClientB, Server\},\{Connector1, Connector2\}, \\
 & & \{(ClientA, Connector1, Server),(ClientB, Connector2, Server)\}, \\
 & & configuration_{nfr}),\ \text{where} \\
configuration_{nfr} & = & \forall\ Conn \in connectors \bullet \\
 & & \text{Compatibility}(Conn_{nfr}, Secure_Communication) \wedge \\
 & & \forall\ Comp \in components \mid IsAServer(Comp) \bullet \\
 & & \quad \text{Compatibility}(Comp_{Pnfr}, Fast) \wedge \\
 & & \quad \text{Compatibility}(Comp_{Pnfr}, Secure)
\end{array}
$$

$IsAServer$ tests if the component is a server (it may be a client) or not. In this case, the constraint $configuration_{nfr}$ defines that every server must be *Fast* and *Secure* and every connector must provide a *Secure_Communication*. The description of the new server is defined as follows

$$NewServer = (\{p_5, p_6\}, Fast \wedge Secure, none, Secure_Communication)$$

Thus, the replacement condition is

$$\begin{aligned} \mathsf{repCompCondition} =& \text{Compatibility}(Secure \wedge Fast \wedge Accurate, Fast \wedge Secure) \wedge \\ & \text{Compatibility}(none, none) \wedge \\ & \text{Compatibility}(Secure_Communication, Secure_Communication) \wedge \\ & configuration_{nfr}(NewServer) \end{aligned}$$

These two servers cannot be replaced, as their non-functional properties are not compatible. Essentially, the $NewServer$ is not *Accurate*, which is a characteristic required by the $ClientA$. As shown on the right hand side of Figure 6, the configuration is not altered, i.e., the old server is not replaced.

6 Conclusion and Future Work

This paper has illustrated how non-functional properties may be explicitly dealt in dynamic distributed systems. In order to perform this task, we use a formal approach and software architecture abstractions for describing dynamic systems. Using a first-order logic based notation and set theory elements, we formally defined NF-Attributes, NF-Requirements, correlation and compatibility notions, components, connectors and configuration. From these definitions, the conditions for executing runtime change operation of adding, removing, replacing and reconfiguring were also formally specified.

This paper presents three main contributions: the treatment of non-functional properties itself, their formalisation and their inclusion in the context of dynamic distributed systems. The formal definition of correlation and compatibility allows the use of tools to perform (semi-)automatic proofs. For instance, a proof may be performed to demonstrate that a system is secure if its non-functional requirements are formally specified. Additionally, it is useful in the identification of conflicts between non-functional properties of architectural elements. In terms of dynamic systems, it creates an actual possibility of including non-functional properties in the checking of the system integrity. None of the current configuration models and architectural infrastructures includes the verification of non-functional properties at runtime. The importance of this treatment was firstly identified in [Oreizy 98a] and is still an important and open research topic within dynamic distributed systems.

In terms of future work, our next target is to incorporate this approach into an implementation environment. Essentially, it is necessary to define how the conditions defined in Section 4 may be included in an actual implementation environment like CL [Justo 99]. On the formal hand side, further investigations have already begun in order to incorporate the runtime treatment of non-functional properties into the ZCL formal framework [Paula 98].

We also intend to apply our approach to other non-functional properties such as *fault-tolerance*, *availability* and so on. Finally, an apparently very interesting point to be considered, refers to the potential benefits of the use of architectural styles [Garlan 95] in relation to the treatment of non-functional properties. Evidences of this fact may be observed because, in opposite to functional properties, non-functional ones usually are general properties belonging to many different software systems. This fact is something very similar to what is desired in the definition of an architectural style, i.e., try to find

similarities among families of software systems and to take benefits of these similarities in the development.

References

[Allen 97] Robert J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1997.

[Boehm 96] Barry Boehm and Hoh In. Identifying Quality Requirements Conflicts. *IEEE Software*, 13(2):25–35, March 1996.

[Chung 99] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-functional Requirements in Software Engineering.* Kluwer Academic Publishers, 1999.

[Filman 98] R. E. Filman. Achieving Ilities. In *Workshop on Compositional Software Architectures*, Monterey, California, USA, January 1998.

[Garlan 95] David Garlan. What is Style? In *Dagshtul Workshop on Software Architecture*, February 1995.

[Ghose 99] Aditya K. Ghose. Managing Requirements Evolution: Formal Support for Functional and Non-functional Requirements. In *International Workshop on the Principles of Software Evolution (IWPSE'99)*, Fukuoka, Japan, July 1999.

[Justo 99] G. R. Ribeiro Justo and P. R. F. Cunha. An Architectural Application Framework for Evolving Distributed Systems. *Journal of Systems Architecture*, 45(1999):1375–1384, 1999.

[Kotonya 98] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Process and Techniques*, chapter 8, pages 190–213. John Wiley & Sons, Inc, 1998.

[Kramer 97] Jeff Kramer and Jeff Magee. Distributed Software Architectures. In *International Conference on Software Engineering (ICSE'97)*, pages 633–634, Boston, USA, 1997.

[Magee 95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architecture. In *Fifth European Software Engineering Conference*, September 1995.

[Medvidovic 00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[Mylopoulos 92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and Using Nonfunctional Requirements: A Process-oriented Approach. *IEEE Transaction of Software Engineering*, 18(6):483–497, June 1992.

[Oreizy 98a] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based Runtime Software Evolution. In *International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[Oreizy 98b] Peyman Oreizy. Issues in Modeling and Analyzing Dynamic Software Architectures. Department of Information and Computer Science, University of California, Irvine, 1998.

[Oreizy 98c] Peyman Oreizy, David S. Rosenblum, and Richard N. Taylor. On the Role of Connectors in Modeling and Implementing Software Architectures. Technical Report 98-04, Department of Information and Computer Science, University of California, Irvine, February 1998.

[Paula 98] Virginia C. C. Paula, George R. R. Justo, and Paulo R. F. Cunha. ZCL: a Formal Framework for Specifying Dynamic Distributed Software Architectures. In *Ninth European Workshop on Dependable Computing (EWDC'9)*, Gdansk, Poland, May 1998.

[Rosa 00] Nelson S. Rosa, George R. R. Justo, and Paulo R. F. Cunha. Incorporating Non-Functional Requirements into Software Architecture. In *Formal Methods for Parallel Programming Theory and Applications (FMPPTA'2000)*, Lecture Notes in Computer Science, 1800, pages 1009 – 1018, Cancun, Mexico, May 2000.

[Rosa 01] Nelson S. Rosa, George R. R. Justo, and Paulo R. F. Cunha. A Framework for Building Non-functional Software Architectures. In *16th ACM Symposium on Applied Computing*, pages 141–147, Las Vegas, USA, March 2001.

[Shaw 96] Mary Shaw and David Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, 1996.