A component-based environment for distributed configurable applications.

Ahmed Saleh[1]
George R. Ribeiro-Justo[2]
Stephen C. Winter[1]

[1] School of Informatics, University of Westminster
[2] Cap Gemini Ernst & Young, UK

This is an electronic version of a paper presented at the 9th IEEE Conference and Workshop on Engineering of Computer-based Systems (ECBS2002): Component-based Software Engineering Workshop: Composing Systems from Components, 08-11 Apr 2002, Lund, Sweden.

9th IEEE Conference and Workshop on Engineering of Computer-based Systems (ECBS2002), Lund, Sweden, April 10 – 11, 2002. Workshop Proceedings.

A Component-based Environment For Distributed Configurable Applications

Ahmed Saleh
University of Westminster, UK
saleha@wmin.ac.uk

George R. Ribeiro-Justo
Cap Gemini Ernst & Young[1], UK
George.Justo@capgemini.co.uk

Stephen C. Winter
University of Westminster, UK
wintersc@wmin.ac.uk

Abstract

One of the basic requirements for distributed applications to run under different working environments is to be flexible, configurable, portable and extensible. Using the current development techniques independently falls short in supporting most of these requirements due to complexity of their integration and the conflict of their objectives. In this context this paper describes an integrated environment based on an interface description language called NCSL, an architecture description language called NADL, and a supporting management system composed of a component-based framework and an event management system that facilitate the process of developing and managing distributed configurable applications based on their non-functional requirements (NFRs).

1. Introduction

While computing power and network technology have improved dramatically over the past decade, the design and implementation of complex distributed configurable applications remain difficult and time-consuming. Also, the need for considering distributed applications' non-functional requirements (i.e. performance, reliability, security, etc.) has added further complexity to the process of developing these applications. Using traditional development techniques often result in static and difficult to understand applications that do not address the user requirements. Also, due to the evolving nature of distributed systems' environments, applications that can tolerate the continuous upgrade of such environments are often developed on a per application basis.

Component-based frameworks have emerged as the new technology that can facilitate the development of distributed applications through reusable components. As its name suggests, a component-based framework is a collection of software components that have been developed independently but can interact and collaborate with each other to support the development of a group of applications or solve a particular type of problems. Unfortunately, constructing distributed configurable applications from pre-existing reusable components of such frameworks cannot be achieved without understanding the structure and functionality of these components. Despite some successful attempts, most of the current frameworks rely on providing reusable components that can be plugged in together in different configurations to build up the applications, but are not able to tackle the problem of design reuse, where the entire structure/architecture of the application can be reused to build new applications. Furthermore, very few frameworks have addressed the problem of integrating the non-functional requirements of the application's services due to the difficulties of representing and controlling such requirements at run-time.

This paper describes an integrated environment for supporting the development and control of distributed configurable applications through a collection of distributed components that collaborate within a specific configuration to satisfy both the developer and environment requirements. This environment is based on a framework of distributed reusable components called FRODICA (Framework for Distributed Configurable Applications). Each constituent component of the framework should have a well-defined interface that has been defined by the NCSL language (Non-functional Component Specification Language) that describes the components' functional and non-functional requirements to enable their interaction regardless to their implementation details. The components' interaction and the configuration itself is defined by an architecture description language called NADL (Non-functional Architecture description Language), which defines the architectural structure of the application and its run-time constraints, and the rules of selecting/integrating different components according to the application's NFRs,.

2. Related Work

Many researchers have investigated the development of component-based frameworks to support the construction of configurable applications in a distributed context. For example, C++CL [1] is an OO (object-oriented) framework for developing reconfigurable distributed systems. It is based on the CL model where an application is divided into two sets of components: tasks and configurations. The computation is usually performed by

[1] The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied of Cap Gemini Ernst & Young.

tasks that can interact with each other via local ports. The configuration is the part of the program where the system structure is specified and controlled. This consists of defining task instances, connecting them and managing their execution. C++CL is considered as a real attempt to create an object-oriented framework for developing dynamic distributed software architectures. However, it does not support the definition of NFRs at any stage of the development process.

The Aster project is another attempt based on matching the NFRs of an application with the NFRs of selected components and connectors manipulated by the Aster framework [3]. This matching process results in generating a customized middleware that provides the NFRs of the application. Although the Aster framework proved to be efficient in implementing several transactional and non-functional properties, it does not cover all concepts of software architecture (e.g. connectors, ports, etc.) only components and some basic connectors are supported. In addition, it does not address the problem of managing NFRs during run time.

Unlike Aster, the QuO (Quality Objects) framework [4] is an integrated environment for developing distributed applications with QoS requirements. Its main idea is based on the notion of *contracts, delegates* and *system condition objects* that negotiate an acceptable region of QoS prior establishing a connection between a client and a server. When both client and server agree upon a specific region, the connection is established and the QoS level is monitored for further developments. Although QuO offers more flexibility than other frameworks, it depends heavily on CORBA IDL to provide its code generator with the appropriate interface, ORB proxy and ORB before generating the executable code of the system. In addition, QuO only concentrates on the structure of the components and their QoS, but does not address the global architecture of the application and its NFRs.

3. The FRODICA Framework

As mentioned in the introduction, the key point to facilitate the development of new applications from pre-existing reusable components is to understand the structure of these components and how they interact. Taking this into consideration, FRODICA [6] has been developed as a four-tier framework that can reside above the operating system and below the application layer. The layering approach adopted by FRODICA categorises the components into four separate layers according to their functionality and complexity. In this context, components of top layers can extend/customise the functionality of the corresponding lower-layer components in order to tailor the topmost-layer components to suit individual distributed applications.

The *communication layer* of FRODICA is the lowest layer of the framework, which is responsible for handling the low-level communication protocols of the system. This layer is mainly concerned with carrying out all the underlying message passing, naming services, binding and data marshalling between distributed components. Accordingly, this layer comprises all platform-dependent software (i.e. libraries and interfaces) required to perform such communications.

The *general-purpose layer* is the middleware layer of the framework that deals with low-level system operations. The main objective of this layer is to hide the platform-specific software and hardware complexity from upper layers, hence provide platform independent environment for system developers to create their applications. This layer acts as the bridge between the application layer and the underlying technology infrastructure. It accommodates a number of management and general-purpose components that provide the basic requirements to build distributed configurable applications.

The *application-oriented layer* is concerned with putting together all the standard services required for supporting the development of an integrated distributed configurable application. Components of this layer are extensively used by system developers in creating their applications, and therefore, they tend to provide the most basic services for developing distributed applications, together with a well-defined interfaces and a clear extensibility methods to enable their use without exploring the complexity of lower layers' components.

Finally, the *specific-application layer* is the topmost layer that comprises the components, connectors and interfaces needed for running a specific application. In this layer, system developers can create their own new components, extend or specialise lower layers' components to build their applications.

4. NCSL Language

The NCSL is a component specification language based on Java. It provides a set of tools for the description and deployment of distributed components, taking into consideration the restrictions and constraints (i.e. non-functional requirements) imposed by the system/developer on these components' services. At the design stage, components are described with the help of a configuration language that defines the internal specifications of each component in terms of the services provided/required by the component, as well as the non-functional requirements associated with each service. The compilation of NCSL into the framework implementation language is achieved via a separate compiler called `NcslToJava`, which examines the validity of the component's interface description and generates an

executable code in the form of Java and XML files. Subsequently, the generated interface will be used by the NADL language (explained at the next section) to identify components' functional and non-functional properties required for configuring distributed applications at run time.

NCSL currently supports three types of non-functional attributes:

- Performance: The performance is defined in terms of average time (measured in millisec) to perform a service.
- Reliability: The reliability is measured in terms of the MTTF (mean time to failures).
- Availability: The availability is measured in terms of the average time to restore (MTTR—mean time to restore) a service after a failure. It is a function of MTTF and MTTR.

The above words are regarded as keywords in NCSL. NCSL also provides the concept of NFR expressions that are Boolean and conditional expressions combining non-functional attribute keywords and their values. For example, a service is required to provide a 'performance = 500 Kb/sec and reliability > 500 mesc'. An example of NCSL illustrated in Figure 1.

```
interface GoldBranch {
                // provided services //
   provide float checkBalance (int customerID,
                                int customer PIN);
       support { performance &&  // supported NFRs
                 availability } ;
                // required services //
   require float getBalance (int customerID,
                             string customer name) ;
  with {performance >= 500 Kb/sec && // required NFRs
            availability >= 500 mesc } ;
   .....
```

Figure 1: The NCSL specifications for a Bank component

To reduce overheads, a component is not required to compute all non-functional attributes, when they are not related to any NFR, but only those critical ones. In this case, NCSL contains a 'support' clause that indicates which non-functional attributes are computed by the component. Remember that the interface corresponds to a contract, therefore if a component supports a non-functional attribute, as described in more details later, the environment and an ADL (Architecture Description Language) script can query the value of that non-functional attribute at runtime.

NCSL adopts the same concepts of ACME (An Architecture Description Interchange Language) [2] in assigning general non-structural information to each architectural entity (i.e. component, connector, port, etc.) to describe its run time behaviour. However, NCSL goes further by defining a set of s to each service supported/required by each one of these entities.

5. NADL Language

Current ADLs allow system developers to integrate heterogeneous software components in a homogeneous way, define and locate distributed components across the network, and adapt their behaviour according to their design preferences. This kind of features is described as the functional requirements of the system. Most ADLs fall short, however, in providing support for the NFRs of the system, which describe its constraints and run-time behaviour. This is due to the fact that they hide the details necessary to specify, measure and control such requirements, and hence provide little support for building systems that can adapt to different levels of QoS. Incorporating NFRs in the design of the system requires the ADL to specify constraints for the QoS properties of the required and provided services of each component. Also, it requires matching techniques for determining whether a service satisfies non-functional requirements and what are the consequences if a component fails to satisfy the desired non-functional requirements.

As a language that supports the description of re-configurable distributed system according to both their functional and non-functional properties, NADL provides special constructs to deal with NFR description and management. An NADL description (Fig 2) is made of two main sections: a configuration section where components are selected according to their services and their NFRs, and a reconfiguration section where reconfiguration actions are taken, depending on the failure or changes of NFRs. NADL also allows the system developer to define environment specific properties that must be satisfied by all components running the application. For example, a component must run within a specific type of operating system or over a machine with certain memory specifications. These properties enable the system developer to refine his selection to identify components that are more specific.

The key constructor of NADL is the concept of NFR expressions that are extensions of those used in NCSL. In NADL, NFR expressions may contain services from different components while in NCSL they refer to the NF attributes of a specific service. For example, the expression below defines that the service *video* provided by component *comp1* should support availability above 500 msec and at the same time, the *sound* provided by component comp2 should perform above 900 Kb/sec:

```
comp1.video.availability >500 msec
&& comp2.sound.performance >900 Kb/sec
```

The NADL selection of components is based on their interfaces, which already specify their NFRs. After the system identifies possible candidates components, the

configuration can be defined. In general, the selection should be the minimum requirement of the system. During the configuration, it is then possible to define further constraints depending on the candidate components that have been selected. In addition, the architect can specify global constraints relating the various components.
The configuration is built by using the typical ADL constructs such as connect, and start. Observe that NADL also uses the concept of default connectors, which are implemented by the supporting middleware. For instance, in the case of Java components communicating using RMI (Remote Method Invocation), it is possible to connect the components directly by using an RMIConnector default connector. After the configuration has been successfully built, the reconfiguration section specifies conditions for monitoring and managing the configuration. This is done using *when* clauses similar to those used during the configuration. The *when* clauses are evaluated sequentially and the first one that satisfies the corresponding reconfiguration block is triggered. During the reconfiguration, components and connectors can be connected or disconnected, and new components and connectors can be selected to satisfy the architecture NFRs.

```
Application : Bank {
 select {
  component: Comp1 { interface: MainBank ;
   location: remote (osiris.cpc.wmin.ac.uk) ;
   properties: { (getBalance.performance >= 500kb/s ||
         checkBalance.availability  >= 5000 msec };};
  connector: Conn1 { interface: GoldConnector ;
   properties: {dataStream.availability >= 800 msec &&
               dataStream.reliability > 750 msec };
                  // End Properties //
            } ; } ;    // End Conn1 // End select //
 constraints: { Comp1.Performance >= 4000 Kb/sec ;
       propertiesCheckupRate >= 4000; }; // Rate of
                              //checking NFRs in msec//
 implementation: {Bank.Platform = java; //App platform
       Bank.OS = Unix;};//OS for running the app //
 configuration: { conf1: when (select) ;
   do (connect Comp1.getBalance To Conn1.dataStream;
      connect Comp3.withdrawCash To Conn2.dataStream);
conf2: when (Comp3.checkBalance.availability <600 ms);
   do (wait (3000);
       reselect;) } ; // Repeat 'select' process
 reconfiguration: {
   when (Comp1.getBalance.performance <5000 Kb/sec ||
         Comp1.getBalance.availability < 5000 msec);
     do ( start ;
            suspend ;
            stop Comp3.checkBalance ;
            stop Comp3.withdrawCash ;
            resume ;
            end) ;
   }; } // End reconfiguration // End Application //
```

Figure 2: The NADL specifications for a Banking Application

NADL also provides the concept of (global) constraints, which define an NFR invariant for the architecture. The constraint is revaluated after every reconfiguration. Observe that, since NADL is service-driven, reconfiguration is carried out at service level, which means that during reconfiguration the whole component is not affected but only those services involved. Further more, component instances offering a service may run longer than a particular application. This means that existing component instances can be shared by different configurations. The architect may decide whether to use a fresh instance of a service or an existing service.

6. Conclusion

The environment outlined in this paper showed how possible it is to extend existing IDLs and ADLs to support the management of NFRs. It has also demonstrated importance of considering the distributed applications' NFRs at the early stages of the design in order to ease their management and control at run-time. Although, we have decided to build our own management service but there is no reason why the management system could not use services of a middleware such as [5], which supports QoS management. We see these two technologies as complementary rather than competing.
Also, the environment outlined in this paper showed that the combination of software architecture with object-oriented frameworks and language mechanisms can lead to the development of a new generation of well-structured distributed applications that can be easily configured to adapt with different working environments.

7. Reference

1. Justo, G. R. R. and Cunha, P.R.F.: "*An Architectural Application Framework for Evolving Distributed Systems*", Journal of Systems Architecture, Special Issues on New Trends in Programming and Execution Models for Parallel Architectures, Heterogeneously Distributed Systems and Mobile Computing, Vol. 45, No. 15, Sep. 1999.
2. Garlan, D., Monroe, R. and Wile, D.: "Acme: An Architecture Description Interchange Language". Proceedings of CASCON, Nov. 1997.
3. Issarny, V. and Bidan, C.: Aster: A CORBA-Based Software Interconnection System Supporting Distributed System Customization. *In Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS'96).* Mayland, USA, May 1996.
4. Loyall, J., Bakken, D., Schantz, R., Zinky, J., Vanegas, R., and Anderson, K.: "QoS Aspect languages and Their Run-time Integration". Proceedings of the 4th Workshop on Languages, Compilers and Run-time Systems for Scalable computers (LCR), Pennsylvania, USA, 1998.
5. Koh, F. and Yamane, T.: Dynamic resource management and automatic configuration of distributed component system. In *Proceedings of the 6th USENIX COOTS*, Jan 01.
6. A. Saleh and G. R. Ribeiro Justo. A configuration-oriented framework for distributed multimedia applications. *In Proceedings of the Fifteenth Symposium on Applied Computing (SAC200)* Italy, ACM Press, March 2000.