Deploying Non-Functional Aspects by Contract

Romulo Cerqueira, Sidney Ansaloni, Orlando Loques
IC/UFF
Niterói, RJ, Brazil
`{curty, ansaloni, loques}@ic.uff.br`

Alexandre Sztajnberg
DICC/IME/UERJ
Rio de Janeiro, RJ, Brazil
`alexszt@ime.uerj.br`

Abstract

This paper presents an approach to describe, deploy and manage component-based applications having dynamic functional and non-functional requirements, which include different types of QoS. The approach is centered on an ADL that allows functional and non-functional requirements to be described by high-level textual contracts. The meta information extracted from the software architecture description is used to guide configuration adaptations required to enforce the ruling contract. The infrastructure required to automatically manage these contracts follows a standard architectural pattern, which can be directly mapped to specific components included in a supporting middleware. This allows designers to write a contract and to follow a standard recipe to insert the extra code required to its enforcement in the supporting middleware. Also, the component configuration capability provided by the middleware helps to put in practice adaptations defined by a contract. Examples demonstrate how the approach and the associated middleware can be used to configure and to support applications with adaptive QoS requirements.

1. Introduction

We present an approach to describe, deploy and manage applications with service guaranties based on the contract concept. This approach defines the relationship between contracts, differentiated quality of services (QoS), and the resource management entities. In our proposal, differentiated quality of services is described by a special language associated to an Architectural Description Language. These descriptions are reflected in the R-RIO framework in two ways: (a) the mechanisms required to automatically manage these contracts follow a standard architectural pattern, which can be directly mapped to specific components included in the supporting middleware; (b) R-RIO connectors can encapsulate a set of different non-functional aspects, and these connectors can be reconfigured at run-time to provide adaptation capabilities.

In the next section we present R-RIO basic concepts. In the sequence we discuss how R-RIO handles the configuration of non-functional aspects in the architectural level using contracts. Then, we illustrate the use of the R-RIO framework with a comprehensive dynamically-adaptable application example. Finally, we present some related works and our concluding remarks.

2. R-RIO Framework

The Reflective-Reconfigurable Interconnectable Objects (R-RIO) framework integrates some key concepts of Software Architecture / Configuration Programming (SA/CP) and Meta-Level Programming (M-LP) approaches [1]. This integration helps to achieve separation of concerns, software reuse and the capability of supporting dynamic configuration. R-RIO framework includes the following elements:

a) A **component model** based on the concepts of SA/CP: (i) **modules**, application components that basically encapsulate functional concerns; (ii) **connectors**, used at the architecture level to define module's interaction relationships. At the operating level, connectors represent, mediate and handle module interaction-domain concerns; (iii) **ports**, identify access points, through which modules and connectors provide or require services, and are also used to explicitly bind modules and connectors.

b) A **software development methodology** that stimulates the designer to comply with a simple meta-level programming discipline, where functional concerns are concentrated in the modules (base level) and non-functional concerns are encapsulated in connectors (meta-level).

c) **CBabel**, an ADL used to describe: (i) application's functional components and interaction topology, (ii) contracts specifying non-functional aspects, and (iii) planned reconfigurations. An architecture described with CBabel can be verified and compiled, resulting in a meta-level architecture description repository used to run and manage the architecture.

d) The **Configurator**, a reflective middleware [2] that provides distributed configuration management and executive services, used to make and control running images from an architecture description.

3. Non-functional aspects

In various application domains we have identified recurring non-functional aspects that should be considered since the design-time: (a) **Interaction,** to configure the characteristics of module interaction; (b) **Distribution,** to configure module location, distribution and communication; (c) **Coordination,** to handle module concurrency and synchronization; and (d) **QoS,** quality of service, regarding operational requirements, such as fault tolerance, security or communication parameters.

To specify non-functional aspects we use the architectural contract concept. In our approach, an architectural contract is a description where two parts express their requirements regarding the application's aspects, and rules for adaptation and negotiation in the presence of context changes.

In our proposal, a functional service is considered a specialized activity, constrained by restrictions that usually do not permit negotiation, defined through the specification of the application's architectural components and their interconnection topology. Non-functional services are defined by restrictions to application non-specific activities that can admit some negotiation on resource utilization. A non-functional contract can describe the use of shared resources an application is going to make and acceptable variations on the availability of those resources at design-time. A contract will be enforced at operation-time by a set of infrastructure components that implement the contract semantics.

In the next section we use QoS domain examples to illustrate our approach to treat non-functional aspects.

4. Examples

We borrowed a subset of the QML QoS language [3] and adapted its syntax and terminology to the software architecture description scope [1]. QoS categories associated to different non-functional aspects are described separately from the components. For instance, if transport performance is critical to an application, a *Transport* QoS category could be described, as shown in figure 1.

```
QoScategory  Transport {
 protocol: enum {UDP, TCP, RTP} ;
 technology: enum {802.11LAN, CDLS, GSM};
 slidingWindowSize: increasing numeric ;
 send_buf-size: increasing numeric ;
 recv_buf_size: increasing numeric ;
 MSS: increasing numeric ;
}
```

Figure 1. Transport QoS category

In a Virtual Terminal application a possible QoS category, defining some of the emulation and security required properties, could be described as in figure 2.

```
QoS category Terminal {
 port: enum {ssh , telnet};
 cipherType: enum {idea, arcfour};
 auth: enum {RSA, password, both};
 connectionAttempts: increasing numeric;
 X11forward: enum {yes, no};
}
```

Figure 2. Terminal QoS category

Figure 3, describes an architecture, that uses a virtual terminal facility, composed by a client and a server module (*vtServer* – lines 2-4, *vtClient* – lines 5-7), and its intended topology (lines 8-9).

```
1   module VirtualTerminal {
2      module {
3         in port (char) Recv;
4      } vtServer;
5      module {
6         out port (char) Send;
7      } vtClient;
8      instantiate vtClient at clientNode;
9      link vtClient to vtServer;
10  } vt;
11  start vt;
```

Figure 3. Virtual terminal functional composition

4.1 Contract with static adaptation

Considering the session establishment aspect, the designer has to select operating parameters related to communication and security. In our example, the first option is to connect to the server using SSH over TCP, and *idea* cryptography with *password* authentication. If this is not possible (eg. the server cannot work with this configuration), a TELNET connection over TCP can be accepted. However, in this case, more *authentication* options and a *receive buffer size* of at least 64 bytes are prescribed.

The CBabel associated contract (figure 4) describing these non-functional requirements specifies two *services*: one for the secure case (lines 2-5) and the other for the unsecured one (lines 6-9). Each of these intended services is fully defined by an associated *profile* that describes the set of required properties, each one associated with a given QoS category; i.e., the profile describes what should be provided by the supporting infrastructure. In the example, the `secTerm` profile (lines 15-20) describes the first operational choice and the `unsecTerm` profile (lines 21-26) the second option. Note that both of them require the TCP protocol. The preferred order for the use of the services is defined in the negotiation clause (lines 11-14). If none of them can be imposed after negotiation the client is unable to open the session. In order to enforce the contract the middleware selects one or more connectors to support the best available service and accordingly configures the architecture. The middleware could configure either a secure or unsecure connector. After the session is established no other adaptation will occur.

```
contract {
  service {
    link vtClient.Send to vtServer.Recv
                  with profile secTerm;
  } secure;
  service {
    link vtClient.Send to vtServer.Recv
                  with profile unsecTerm;
  } unsecure;
  negotiation {
    secure   -> unsecure;
    unsecure -> out_of_service;
  }
} vt;
profile {
   Transport.protocol: TCP;
   Terminal.port: ssh;
   Terminal.cipherType: idea;
   Terminal.auth: password;
} secTerm;
profile {
   Transport.protocol: TCP;
   Transport.recv_buf_size: > 64;
   Terminal.port: telnet;
   Terminal.auth: both;
} unsecTerm;
```

Figure 4. Virtual terminal QoS contract

4.2 Contract with dynamic adaptation

In a different use-case scenario, a sales-person has to visit potential consumers and be permanently connected to the main office database through a virtual terminal. She uses a mobile device that can run a virtual terminal client and provides communication with three possible operating modes:

- in the range of a cordless station base, it can operate over a regular wired telephone line;
- on the move, it can detect a cellular antenna and reconfigure to use a cellular connection;
- when immerse in a wireless network, it can use the available communication protocols.

For each operating mode there is an appropriate communication interface and a sensor that detects if the associated channel is available or not. While in operation the supporting middleware tries to establish a link using the best available channel. Figure 5 presents the application's overall architecture. The dashed lines represent the links that can be dynamically established, guided by the negotiation policy and depending on resource availability.

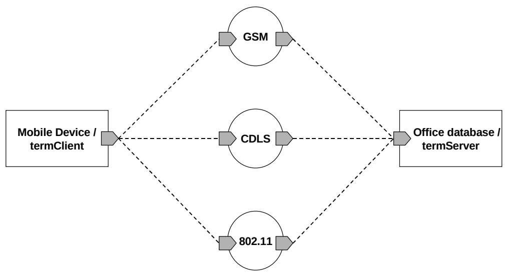

Figure 5. Mobile device basic architecture

The contract for this use-case (figure 6) defines three different *services*, each one associated to a different operating mode supported by the device. The support for each service is encapsulated in a connector. The *negotiation* clause (lines 14-18) defines that the best service is the `wireless` transport service. If this service is not available, a `fixTel` service should be tried and so on. Transitions between services depend on their availability. Either when the currently used link fails, or a preferable service becomes available, a service adaptation can take place. Other adaptation policies could be used, e.g., based on a reduced-cost or bandwidth criterion.

```
1  contract {
2    service {
3       link vtClient.Send to vtServer.Recv
4       with Transport.technology: 802.11LAN;
5    } wireless;
6    service {
7       link vtClient.Send to vtServer.Recv
8       with Transport.technology: CDLS;
9    } fixTel;
10   service {
11      link vtClient.Send to vtServer.Recv
12      with Transport.technology: GSM;
13   } celTel;
14   negotiation {
15       wireless -> fixTel;
16       fixTel   -> celTel;
17       celTel   -> out_of_service;
18   };
19 } vt;
```

Figure 6. Mobile device QoS contract

4.3 Composing Contracts

Contracts regarding different non-functional aspects (in the same or in different applications) can be orthogonal and cause no interference with each other. In this case, composing those contracts is immediate. In the general case, the composition process can lead to conflicts on the use of shared resources. In our proposal, the composition of contracts can be specified combining in a unique clause the *negotiation* clauses of the involved contracts. Conflicts could be handled assigning priorities to each of the composed contracts.

Regarding our example combining the two contracts is trivial. Services and profiles would be part of the same contract and the unified negotiation clause would be described as in figure 7. We can note that both contracts are orthogonal, given that it is not necessary to combine states of the two sets of services in the same negotiation chain.

```
negotiation {
   secure   -> unsecure;
   unsecure -> out_of_service;

   wireless -> fixTel;
   fixTel   -> celTel;
   celTel   -> out_of_service;
}
```

Figure 7. Combining contracts

4.4 Support

CBabel described architectures, functional composition and non-functional contracts, are stored as meta-level information (figure 8). With this information a set of middleware components, arranged in a well-defined architectural pattern [4], will interact with the Configurator (section 2) to instantiate and bind application components, and enforce the contracts:

Contract Manager interprets contract descriptions to extract the service negotiation state machine. When a negotiation is initiated, the Contract Manager identifies which service will be negotiated and sends the related configuration descriptions and the associated profiles to the Interactor. If every service inside a negotiation clause has been unsuccessfully tried, an *out-of-service* state is reached, and a contract violation message is returned to the user. As seen in the example, the contract manager can also start a negotiation when a preferable service becomes available.

The **Interactor** has some responsibilities: (a) translate service profiles properties into system-level support services and request those services with adequate parameters to the QoS Agent in order to *instantiate* / *start* modules with required QoS, (b) map the service *link* interaction information into a connector able to provide the required interaction QoS, (c) call the Configurator to actually perform architectural procedures (instantiate, link, start) on modules and connectors with the configured resource context, and (d) receive *out-of-range* notifications from the QoS Agent. When this occurs, based on its internal programming the Interactor can try to readapt the resource allocation, at resource-level (for instance changing resource parameters, if possible), or send an *out-of-service* notification the Contract Manager to initiate another architecture-level negotiation action.

QoS Agents encapsulate the access to system level mechanisms. Their main responsibilities are to make the actual resource allocations, initiate local system services and to monitor required property values. According to the monitoring thresholds registered by the Interactor, the Agent can send back an *out-of-range* property notification.

Regarding our examples, the QoS Agent is responsible for verifying the availability of the virtual terminal properties (telnet or ssh ports, authentication, etc.) before establishing a terminal session. The QoS Agent also encapsulates the sensors functions, monitoring the transport channel operation. The Interactor is programmed to instantiate a combination of connectors with suitable parameters and to ask the QoS Agent to start some resource reservation procedures to maintain transport and security options for each service. Also, it registers with the QoS Agent the interest for new channel-available events. When a negotiation occurs (say, change from `wireless` to `fixTel`) the Contract Manager sends another service request to the Interactor that, by its turn, selects a connector encapsulating the required transport technology mechanism, and invokes the Configurator to dynamically change the architecture's composition.

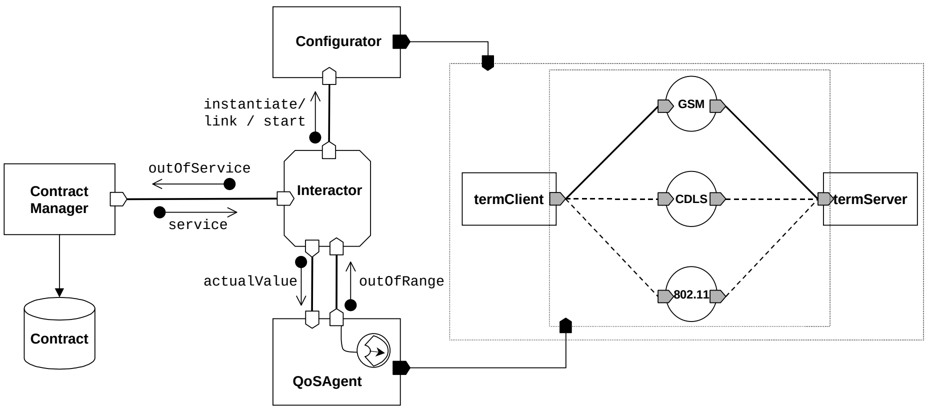

Figure 8. QoS supporting component infrastructure

5. Related work

QML [3], which was the main inspiration of our contract description language, is applied in the class-object context, and is not directly applicable to the implementation level. Quartz [5] provides an API to describe QoS requirements, through a set of parameters that are used to select components of the supporting middleware. The approach seems to be restricted to multimedia QoS requirements and relies on services particular to CORBA infrastructures. Our approach has similarities with the proposals presented in [6] and [7], which are in fact interrelated. Most of the elements of the pattern language describe in [6] have a counterpart in our proposal. For example, their Quality Connector performs functions similar to those provided by our Interactor and QoS Agent combined. [7] proposes a contract pattern that could be used to provide part of the support for dynamic adaptations required in our middleware. Our proposal provides an integrated framework to deploy QoS and other non-functional aspects, starting to handle these concerns from the architectural description. In addition, using our configuration programming environment, and the embedded architectural meta-level information, we can handle adaptations through component (re) configurations in a natural fashion.

6. Conclusion

We presented a unified approach to specify, deploy and manage applications having non-functional requirements. The approach allows non-functional aspects of an application, such as QoS requirements, to be specified using high-level contracts expressed in an extended ADL. Being centered on an ADL-based configuration middleware the framework inherits all its well-known benefits, among them the capability of reconfiguration, which helps to execute dynamic architectural adaptations in behalf of a contract.

Part of the codification related to a non-functional aspect can be encapsulated in connectors, that can be (re)configured at run-time in order to cater for the impositions defined by the associated contract. Also, the infrastructure required to enforce the contracts follows an architectural pattern that can be implemented by a standard set of components of the middleware. In this pattern, each component performs a well-defined role in the support of the contract. We have evaluated the approach through several case studies that showed that the code of these supporting components may change partly according to the specific contract. However, we should notice that the treatment of low-level details always has to be considered in any QoS aware application. We believe that our approach can help to identify the intervening points and make the required adaptations more rapidly.

We have used the described approach to specify a reasonable number of contracts, defined at the architecture description level. These contracts have been mapped to implementations using the pattern described in this paper. Through this activity, we have identified some recurrent structures inside the components of the pattern. We think that on making these structures explicit, and available to designers, the task of mapping architecture-level defined contracts to implementations can be simplified.

Acknowledgments

Romulo Cerqueira is supported by Petrobras. Orlando Loques and Alexandre Sztajnberg are partially funded by CNPq (grants PDPG-TI 552137/2002 and 552192/2002, respectively).

7. References

[1] Loques, O., Sztajnberg, A., Leite, J. and Lobosco, M., "On the Integration of Meta-Level Programming and Configuration Programming", Reflection and Software Engineering, LNCS V. 1826, pp.191-210, June, 2000.

[2] Sztajnberg, A. and Loques, O., "Reflection in the R-RIO Environment", Workshop on Reflective Middleware, Palisades, NY, April, 2000.

[3] Frolund, S. and Koistinen, J., "Quality-of-service specifications in distributed object systems", Distributed Systems Engineering, IEE, No. 5, pp. 179-202, UK, 1998.

[4] Carvalho, S. T, Lisbôa, J. and Loques, O, "A Software Architecture Configuration Design Pattern", 2nd Latin American Conference on Pattern Languages of Programming, Itaipava, RJ, Brazil, August, 2002.

[5] Siqueira F. Cahill, V., "Quartz: A QoS Architecture for Open Systems", *18º Simpósio Brasileiro de Redes de Computadores*, pp. 553-568, Belo Horizonte, MG, Brazil, May, 2000.

[6] Cross, J. K and Schmidt, D. ,"Quality Connector – A Pattern Language for Provisioning and Managing Quality-Constrained Services in Distributed Real-Time and Embedded Systems", 9th Conf. on Pattern Language of Programs, Monticello, Illinois, September, 2002.

[7] Zinky, J. A., Bakken, D. E., Schantz, R. E., "Architectural Support for Quality of Service for CORBA Objects", *Theory and Practice of Object Systems*, John Wiley & Sons, Inc., Vol. 3, No. 1, 1997.