R-RIO Framework

Alexandre Sztajnberg
Dep. de Informática e Ciências da Computação, IME/UERJ
Rio de Janeiro, RJ, Brasil
alexszt@ime.uerj.br

Orlando Loques
Instituto de Computação, UFF
Niterói, RJ, Brazil
loques@ic.uff.br

ABSTRACT

Separation of concerns is a key goal in achieving software reusability. Meta-Level Programming approaches pave the way to separation of concerns by handling functional and non-functional aspects in different levels, but provide little help for software composition, verification and evolution activities. Approaches based on Software Architecture / Configuration Programming can overcome these deficiencies and additionally may discipline, and make explicit, the deployment of meta-level programming. Our proposal combines both approaches providing a useful framework to develop, implement and maintain applications.

Keywords: separation of concerns, software architecture, reflection, quality of service

1. Problem Description

Modern computer applications must be developed rapidly in order to meet market demands. Variants of a basic functional system have to be delivered in a short time, and comply with specific functional and non-functional requirements. Successful software development for those applications would benefit from some common guidelines:

- modularity is a prime to collect a functional component set and selectively add to it non-functional features; modularity will help reusability if carefully applied;
- components can be independently designed, may be implemented using different programming languages, and run on different operating platforms;
- applications may have to change their component makeup during theirs life-cycle; software architectures have to be flexible to evolve dynamically;

In this context, methodologies, development systems and supporting environments that can integrate these guidelines into systematic software-engineering practice, are necessary. Compositional development and separation of concerns, with which different requirements can be tackled separately, are key concepts to attain this goal.

Meta-Level Programming (M-LP) approaches allow arranging software elements in different levels of concerns. Using reflection-like techniques, the designer can isolate non-functional (including operational) requirement handling in a meta-level and have the base level computation reified to that meta-level whenever necessary. In addition, reflection allows applications to reason about themselves and possibly make adaptations, in face of operational status changes. However, M-LP is usually associated with specific object-oriented languages, where composition is achieved using inheritance mechanisms that hide the actual structure of the software inside the objects. This software structure opacity makes verification and dynamic evolution activities fairly difficult.

Software Architecture / Configuration Programming (SA/CP) allows the description of software systems in an abstract level, and explicitly separating concerns regarding functional components from their interaction schemes. This makes it easier for the designer to understand the system architecture and to configure applications to fulfill specific

requirements. SA/CP are described with Architecture Description Languages (ADL), which are suitable for property and architectural conformance checking due to the explicit module composition exposition. This also helps achieving a natural mapping from described SA to the actual system software structure, and can in a later stage facilitate dynamic reconfiguration activities.

Despite of presenting some appropriate features for application development, when taken separately, the use of SA/CP and M-LP have some drawbacks. M-LP tools and reflection run-time support environments do not provide direct support for reconfigurations activities. In this case, reconfigurations are usually programmed in ad-hoc manner and rely mostly on the programmer's skills, hindering reuse. Property verification is also more difficult because the software structure is not exposed. In SA/CP proposals the explicit description of non-functional aspects is not a common practice. Most contemporary proposals only consider specific non-functional aspects, such as remote communication.

Combining M-LP and SA/CP, by handling component interconnection and interaction (non-functional) concerns in a meta-level can provide a framework that encompasses the advantages of both approaches. In this way, SA/CP can discipline the use of reflection by providing a simple and systematic mechanism to reify interactions. In addition, dynamic changes in the application, triggered from a meta-level, can now be handled by the SA/CP software reconfiguration capability [2]. Our goal is to demonstrate that this combination can provide a sound framework to develop the intended class of applications.

In the next section we introduce the R-RIO framework. Following, we discuss how R-RIO handles the configuration of non-functional aspects, such as Quality of Service (QoS) in the architectural level. Finally we present our last observations regarding related works and concluding remarks.

2. R-RIO Framework

R-RIO (Reflective-Reconfigurable Interconnectable Objects) is a framework that aims to achieve some directions to guide the construction of tools to design, build, verify, operate and maintain software architectures. R-RIO integrates some key concepts of SA/CP and M-LP approaches [2]. This integration helps to obtain separation of concerns and improve software reuse. In addition, the capability of supporting dynamic configuration and flexibility on component programming language choice are potentially improved. R-RIO includes the following elements:

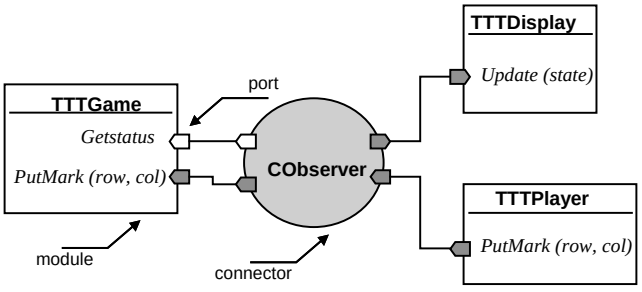

Figure 1 – TicTacToe game with R-RIO

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 relationships. At the operating level, connectors encapsulate, mediate and handle module interaction-domain concerns and interaction with the operating support; (iii) **ports**, identify access points (through which modules and connectors

provide or require services) and are also used to explicitly link modules and connectors. Figure 1 depicts a simple architecture modeling the TicTacToe game [4].

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

c) A **configuration model** that allows for the dynamic creation, connection, deletion and reconfiguration of components and applications.

d) **CBabel**, an ADL used to describe: (i) application's components and interaction structure, (ii) contracts specifying non-functional concerns such as special interaction patterns and, in addition to other ADLs [8], coordination, distribution and QoS, and (iii) planned reconfigurations. Further, an architecture description can be verified and integrated to a running configuration as meta-level information. A short description of the TicTacToe example architecture would be:

```
module  TicTacToe  {
   instantiate   Game, Player, Display;
   instantiate   CObserver;
   link Player, Display to Game by CObserver;
}
module  TicTacToe  TTT;
instantiate  TTT;
start TTT;
```

e) A **reflective middleware** that provides a configuration management API, used to make and control running images from a software architecture description, and an architectural reflection API, that can be used by a running application to collect meta-level information, kept by the middleware, and reason about its own architecture in order to perform reconfigurations [2, 3]. This middleware also facilitates implementing dynamic adaptation and evolution activities.

3. Mapping

The mapping of modules, ports and connectors to real software artifacts depends on the particular adopted environment. In our prototype, primitive module types are defined by Java classes, and composite modules can be composed by arbitrary configurations of primitive modules. Ports are associated to Java methods declarations (signatures) at the configuration level and to method invocations at the code level. It is important to note that only the methods explicitly associated with ports are configurable through connectors and directly visible at the architectural level. Connector types are currently defined and composed as modules, but they have a special implementation and are specially treated by the configuration management. Module and connector types (mapped to Java classes) are associated to module and connector instances through R-RIO's ADL declarations. At configuration time, module and connector instances are created as Java objects. This mapping helps to maintain separation of concerns allowing reconfiguration of every described component in the architecture level during run-time.

4. Non-functional aspects

In the R-RIO approach, the concept of contracts is used to specify non-functional aspects in the architecture level. During our researches we have identified recurring non-functional aspects used in various application domain and, additionally, we verified that this set of aspects could be adequately encapsulated in connectors. In that way R-RIO handles the following non-functional aspects:

- **Interaction –** configure interaction styles and features between modules;
- **Distribution –** module location is independent of its mapping / implementation (here are included reference maintenance and communication mechanisms);
- **Coordination –** including the handling of internal and interaction related module concurrency and synchronization control;
- **QoS –** quality of service aspects, regarding operational requirements, such as fault tolerance, temporal restrictions or communication protocol parameters.

In R-RIO, each of those non-functional aspects can be described with CBabel contracts. Therefore, a given software architecture can be seen as a functional description (including module interface and composition topology) and multiple non-functional contracts.

One can observe that the mentioned aspects can be considered, in principle, orthogonal to the module definition. This facilitates the description of a non-functional aspect and its encapsulation in connectors. Nevertheless, specific non-functional aspects can be required on an application (for instance, consider multimedia flow synchronization) or in some applications might not be possible to completely separate every aspect (consider exception handling). In that case, the designer is not forbidden to use other non-functional aspects or to aggregate functional and non-functional aspects in the same module, but this should be avoided.

In the next section we will discuss how a QoS requirement can be included in the TicTacToe application with the R-RIO approach.

5. QoS contracts in CBabel

To describe QoS aspects within CBabel we borrowed a subset of the QML language [5] and adapted its structure and terminology to an architecture description scope [1]. QoS *categories* regarding an operational aspect are described separately form the components. A module can have a QoS *profile* clause, describing how a given QoS category is required or provided. For instance, if the *Game* module is critical, a *Replication* QoS category could be described, as shown next, and a QoS profile should be associated to the modules.

```
QoS category  Replication {
   numberOfReplicas: increasing numeric ;
   replicationPolicy: enum {PassiveColdStandBy, PassiveHotStandBy, SimpleActive}
   fltTol: increasing enum {restrict, byzantine} with order {restrict < byzantine }
}
```

A QoS contract can be encapsulated in a QoS connector, relating provided and required QoS profiles associated with the involved modules. There can also be a violation policy handling associated with a contract (e.g., *renegotiate* and *terminate*). A simple QoS contract for the TicTacToe example can is described in a *QoS connector* as follows:

```
connector QoSCObserver {
  QOScontracts {
    (Player, Game) renegotiate;
    (Display, Game) terminate;
} }
```

Once an application's QoS aware architecture is described with CBabel it is possible to submit it to coherence and consistency checking (for instance, a client should not require a QoS that a server cannot provide). This allows the designer to reason about the QoS prior to any other step towards implementation.

5.1 Support

Given a CBabel architecture description containing QoS contracts, a standard recipe maps QoS descriptions into actions and data that are stored as meta-level information. R-RIO reflective middleware interprets this information to configure the adequate resources and initiate appropriate procedures to enforce the QoS. To this end, the application has to provide two kinds of special modules. *QoS Configurators* will interpret QoS descriptions stored in the meta-level and compose a QoS connector that will enforce QoS. The QoS Configurator will be invoked at binding time. *QoS Monitors* are special modules that will continuously measure the actual QoS parameters and check if they are in the required range. According to the violation handling policy described in the contract, the QoS Monitor can ask the QoS Configurator to reconfigure the QoS connector, if a violation occurs.

As QoS contracts in CBabel are not restricted to multimedia, a diverse infrastructure is necessary to enforce every QoS category. For instance, the resources required to assure reliability are different from those required to assure performance. The main advantage of the R-RIO's approach to QoS is the mapping mechanism that can be used to handle any QoS category, and the infrastructure composed by the configuration service middleware, QoS Configurator and Monitor can be designed as a *pattern*. It is up to the QoS Configurators and QoS Monitors (selected by the application programmer) to select an adequate QoS-connector composition to actually enforce QoS. New QoS categories can be handled, as specialized QoS Configurators and QoS Monitors are available. R-RIO will not provide QoS enforcement mechanisms, but will encapsulate existing ones in connectors. In the given example, if QoS were not separate from the base functionality, the involved modules would have to encapsulate fault-tolerance enforcement code even if it would not be used immediately. Our technique uses meta-level component selection (connectors) and composition to map QoS described with CBabel into a deployable implementation [1].

6. Related Work

The R-RIO approach to application design was compared to related proposals. These proposals were separated in two groups: M-LP and SA related. Regarding the first group we examined aspect oriented programming (*AspectJ*, *Configuration Filters*), reflective programming languages / environments (*MetaXa*, *Open C++*) and reflective middleware approaches (*2K*, *QuO*, *Quartz*). Regarding the second group we investigated other ADLs and configuration environments (*Regis/Darwin*, *UniCon*, *C2*). Variants on the producer-consumer with bounded-buffer example were developed as a common suit to be used in our evaluation for each case (details and references to the mentioned and other proposals can be found in [6]). We found out that R-RIO has equivalent expression power and demands comparable programming effort, and can be more flexible regarding other approaches, given that beneficial features included in both groups are present in a single framework. For instance, with CBabel the application's architecture is explicitly described in its functional and non-functional aspects. This description is available for consulting during running time (architectural reflection). R-RIO also provides basic support for reconfiguration activities regarding this same architecture. In other proposals this support is partially provided and, when available, the programming is often *ad hoc*.

7. Experiences using R-RIO

Beyond the producer-consumer suit we developed a set of examples using the R-RIO framework where non-functional aspects such as interaction patterns (distributed *TicTacToe*), coordination and assisted reconfiguration (*Beer Factory Plant*) and, QoS with dynamic

reconfiguration (the *3-in-1 Phone*) could be exploited [6]. Additionally we performed some measures to verify R-RIO connector performance and overhead in module interaction [7]. These experiences allowed us to improve R-RIO concepts.

8. Concluding Remarks

The integration of SA/CP and M-LP approaches in the R-RIO framework facilitates the practice of separation of concerns, reuse and application's dynamic evolution. R-RIO allows for a flexible configuration of functional and non-functional aspects. Considering QoS in the architecture level also facilitates the deployment of operational aspects.

The R-RIO's component and configuration models are quite stable. We also developed a prototype for the reflective middleware. One of its features, that improves the flexibility of our approach, is the support for context-reflective adaptation, i.e., generic connectors - encapsulating specific concern-related code and off-the-shelf communication mechanisms - can be automatically and dynamically adapted to any module interface signature [3]. A CBabel compiler and prototype GUI were also developed. The compiler can perform structural verification and integrate CBabel architecture descriptions with the configuration middleware. The GUI, is actually a browser integrated with the compiler, with which one can graphically design and manage (running) software architectures.

Currently we are improving our tools and developing examples of different domain with QoS requirements. We are also working on a formal approach to describe and verify R-RIO software architectures.

8. References

[1] Sztajnberg, A. and Loques, O., "Bringing QoS to the Architectural Level", ECOOP 2000 Workshop on QoS on Distributed Object Systems, Cannes France, June, 2000.

[2] Loques, O., Sztajnberg, A., Leite, J. and Lobosco, M., "On the Integration of Meta-Level Programming and Configuration Programming", In Reflection and Software Engineering (special edition), V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

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

[4] Sztajnberg, A., Lobosco, M. and Loques, O. G., "Configuring Interaction Protocols with the R-RIO approach ", In Proceedings of the Brazilian Symposium on Software Engineering 99, Florianópolis, SC, Brazil, October, 1999. (Portuguese)

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

[6] Sztajnberg, A., *Flexibilidade e Separação de Interesses para a Concepção e Evolução de Aplicações Distribuídas*, D.Sc. thesis proposal, PEE/ COPPE/UFRJ, Rio de Janeiro, July, 1999. (Portuguese)

[7] Sztajnberg, A., *Medidas de desempenho de conectores em R-RIO*, Tech. Report R-RIO-002, GTA/COPPE/UFRJ, 2000. (Portuguese)

[8] Medvidovic, N., Taylor, R. N., "A Framework for Classifying and Comparing Architecture Description Languages", *6th European Software Engineering Conference*, Zurich, Switzerland, September, 1997.