Reflection in the R-RIO Configuration Programming Environment

Alexandre Sztajnberg (UERJ - alexszt@uerj.br) and **Orlando Loques** (IC/UFF - loques@ic.uff.br)

1 Introduction

R-RIO (Reflective-Reconfigurable Interconnectable Objects) is a distributed programming environment that integrates some key concepts of Configuration (CP) and Reflective Programming (RP) approaches [1]. This integration helps to achieve 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. In this position paper, we highlight how reflection is used in three distinct fashions in R-RIO. In a first form, reflection is used to reason about application architectures built with the support of R-RIO. In this form, one can instantiate applications with customized architectures and dynamically adapt their architectures to new requirements. In a second form, computational reflection is used through connectors to allow the addition of non-strictly-functional aspects, such as distribution, coordination, fault-tolerance and QoS support to applications. In a third form, structural reflection is used to dynamically adapt connectors to the context of the functional software module instances for which they have to mediate interactions, augmenting the reuse and easing off-the-shelf middleware adoption.

2 R-RIO Details

The R-RIO approach is based on: (i) a design methodology that relies on composition of functional components and non-strictly-functional connecting elements to build application architectures; (ii) an architecture description language (ADL), called CBabel, with witch are described the component selections, interconnections and composite elements that will be used to structure the applications, and non-strictly-functional aspects such as coordination, distribution, fault tolerance or QoS that will allow to customize those applications; and (iii) a supporting middleware that allows managing application architectures.

R-RIO's software model includes three basic elements: (a) **modules**, application components that basically encapsulate its functionality; (b) **connectors**, used at the configuration level to define module's interaction relationships. At the operation level, connectors mediate and handle module interactions; (c) **ports**, identify access points (through which modules and connectors provide or require services) that are used to link explicitly modules and connectors. Port names provide the elements of liaison between the configuration descriptions and module level interaction mechanisms of specific programming environments.

In our prototype environment, 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. Connector types are currently defined and composed as modules, but they have a special implementation and are specially treated by the configuration tools. 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.

R-RIO includes a configuration management service that is used to create running images and to reconfigure already executing application architectures. The configuration management service has a layered structure that is implemented by two system components: (i) an executive, that provides basic methods for configuration control: module and connector instantiation/deletion as well as binding/unbinding of modules and connectors; and (ii) a manager, that provides higher level functions, such as, interpreting configuration description scripts, answering to configuration queries and monitoring configuration status and events.

3 Kinds of Reflection in R-RIO

R-RIO uses reflection in three different forms: for the dynamic management of software architectures; for the customization of software architectures through selection of connectors; and for the dynamic adaptation of existing generic connectors to mediate interactions of modules having different interface signatures. The following table summarizes the correspondences between reflective programming abstractions and those of the R-RIO's model, where reflection is disciplined by the use of configuration programming concepts.

Reflective Programming		R-RIO
base-level components	↔	*(configurable)* modules
reification points	↔	*(linkable)* ports
meta-level components	↔	*(generic / configurable)* connectors

Architectural Reflection: A running architecture can be configured and monitored, from a meta-level (using the configuration manager), in order to reflect on its current state and dynamically alter its module/connector structure, in face of events that change the current application requirements. For planned reconfigurations, we separate decision-making policies from reconfiguration actions through a simple ADL construct that is used to specify the name and scope of configuration management scripts. The execution of these configuration management scripts can be externally triggered by invoking a standard method on the configurator. Other researchers propose extended ADLs with capability of expressing configuration monitoring mechanisms and management policies separately, which are defined by special ADL constructs. Although similar capabilities could have been added to CBabel (R-RIO's ADL), we opted for keeping our ADL concise, introducing these concerns precisely where necessary.

Architecture Customization: Computational reflection is used in R-RIO disciplined by the use of connectors. Specific connectors can be selected, through configuration, to add new non-strictly-functional aspects to the application. When implementing a connector a programmer can access information associated to method invocations (associated to ports), as they were intercepted and reified to the meta-level (implemented by the connector), and perform some work before returning to the base level. In this way, information exchanged through connectors can be inspected and manipulated and specific interaction styles can be imposed. For instance, one can configure *logical guards* into a connector to add coordination restrictions to method invocations used for the interaction of modules.

Context-Reflective Connectors: R-RIO's connectors are context-reflective in the sense that, when plugged into modules, they can be automatically and dynamically (at configuration time) adapted to mediate interactions between ports using any method signature, including the exception definitions associated to each particular method. This capability is achieved using the structural reflection capability of Java to adapt, at run-time, the particular connector to the context of signatures defined by the set of modules for which it is mediating interactions. Using this approach it is not required to statically generate and keep in a repository either stubs or connector executable code for each specific pair of connectable ports. We took advantage of this capability to provide generic connectors to support common communication styles (Java RMI, sockets, multicast, etc) used in distributed applications. This same technique, can be used to make generic connectors to encapsulate mechanisms to support functions such as, e.g., encryption / decryption, compression / decompression, state saving, transactions support and fault-tolerance. We also are investigating the use of this automatic adaptation capability to provide generic connectors to encapsulate interaction-related design patterns such as: chain of responsibility, visitor, memento, mediator and observer.

The structure of R-RIO's connectors follows a standard design pattern and can be programmed according to a well-defined recipe, and which implementation can be supported by a tool. Basically, at the invoker side, each method invocation is serialized and, in the sequel, at the target object side a dynamic invocation of the concerned method is performed. In a distributed environment, the serialized data is transmitted over the network. It is worth to point out that our connectors are implemented on the standard Java virtual machine. We have not put much effort in optimizing our connector's implementation. However, we made a set of measurements that show a small overhead when compared to the cost associated with the basic machinery required to perform dynamic method invocations in Java, which is considerably greater than the normal method invocation cost.

4 Conclusion

In R-RIO, meta-level components can be mapped into connectors and application customization can be achieved by specifying connectors to compose architectures through configuration programming. The static and dynamic configuration capability of modules and connectors simplify architecture composition and evolution, through the execution of planned and unplanned architectural changes. In addition, the availability of context reflective connectors may enable the synthesis of highly dynamic architectures as those made possible in the WWW context.

Currently we are extending our ADL to allow descriptions of QoS specifications (behavioral contracts). This would lead us to two new interesting and complementary capabilities: (i) to verify beforehand if functionally compatible modules will obey QoS interaction contracts in a given distributed system architecture; (ii) and to assemble composite connectors, which may include the configuration of specific support components, required to meet the specified QoS contracts. A preliminary version of R-RIO is available for research use; see http://www.ic.uff.br/~rrio for details.

Acknowledgments: Alexandre Sztajnberg is supported in part by a grant from CAPES-Brazil, BEX 1014/99-4.

Reference

1. O.G. Loques, A. Sztajnberg, J.C.B. Leite e M. Lobosco, "On the Integration of Configuration and Meta-Level Programming Approaches", RT-02/2000, IC-UFF, Brazil, http://www.ic.uff.br/reltec.html.