R-RIO: Reflective-Reconfigurable Interconnectable Objects

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

Orlando Loques
CAA/IC/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 Description Languages can overcome these deficiencies and additionally may discipline, and make explicit, the deployment of meta-level programming. R-RIO combines both approaches providing a useful framework to develop, implement and maintain applications.

1 Introduction

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 guide-lines:

- 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;
- verification of formal properties should assure the quality of the delivered software.

In this context, methodologies, development systems and supporting environments that can integrate these guide-lines 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 (MLP) approaches allow arranging software elements in different levels of concerns. Using reflection-like techniques, the designer can isolate non-functional requirement code in a meta-level and have the base level computation reified to that meta-level whenever necessary. However, MLP is usually associated with specific object-oriented languages, were composition is achieved using inheritance mechanisms, that hide the actual structure of the software inside the objects. The resulting software structure opacity makes verification and dynamic evolution activities fairly difficult.

Component-Based (CB) approaches allow composing applications from heterogeneously-built components, providing reusability and interoperability. Some CB development environments provide mechanisms for non-functional requirement programming (e.g., CORBA interceptors), but they lack adequate concepts and mechanisms to describe, configure (non-functional aspects included), and perform formal verification of applications in a systematic way.

2 Project Goal

Our goal is to demonstrate that the combination of the abstraction, configuration description, and analysis capabilities of Software Architecture/Configuration Programming (SA/CP) approaches, and the reification flexibility provided by MLP (including its intrinsic separation of concerns support capability) can provide a sound framework to develop the intended class of applications.

SA/CP goes some steps further from CB, allowing 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 overall architecture and to configure applications to fulfill specific non-functional 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 the described SA to the actual system software structure, and can in a later stage facilitate dynamic reconfiguration activities.

MLP techniques, such as reflection, by their turn, allow us to design applications in separate levels: a base level, were basic functional computation is performed and a meta-level, were non-functional (including operational) computation can be handled. In addition to encapsulating non-functional concerns in a meta-level, reflection allows applications to reason about themselves and possibly make adaptations in their software composition, e.g., in order to adapt to operational status changes.

Despite of presenting some appropriate features for application development, when taken separately, the use of SA/CP and MLP have some drawbacks. MLP 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,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA 2000 Companion Minneapolis, Minnesota
© Copyright ACM 2000 1-58113-307-3/00/10...$5.00

hindering reuse. Property verification is also more difficult because the software structure is not exposed.

The explicit description of non-functional aspects in SA/CP proposals is not a common practice. Most contemporary proposals only consider specific non-functional aspects, such as remote communication. Combining MLP 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 mechanism to reify interactions. In addition, dynamic changes in the application, triggered from a meta-level, can now be handled in a systematic way.

3 R-RIO

R-RIO (Reflective-Reconfigurable Interconnectable Objects) integrates in single framework key concepts of Software Architecture / Architecture Description Languages (SA/ADL) and Meta-Level (MLP) Programming 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 the following, the main elements of R-RIO are presented.

- A component model based on the concepts of SA/CP: (a) **modules**, application components that basically encapsulate functional concerns; (b) **connectors**, used at the architecture level to define module's interaction relationships. At the operating level, connectors encapsulate, mediate and handle module interaction-domain concerns; (c) **ports**, identify access points (through which modules and connectors provide or require services) and are also used to link explicitly modules and connectors.
- A software development methodology that stimulates the designers to comply with a simple meta-level programming discipline, were functional concerns are concentrated in the modules (base level) and non-functional concerns are encapsulated in connectors (meta-level) [2].
- A configuration model that allows for the dynamic creation, connection, deletion and reconfiguration of components of an application architecture.
- CBabel, an ADL used to describe: (i) application's components and interaction structure; (ii) contracts specifying non-functional concerns (such as coordination, distribution, QoS, and special interaction patterns); and (iii) planned reconfigurations.
- A reflective middleware that provides configuration management and executive services, used to make and control running images from a software architecture. Through architecture-level reflection, an application can collect meta-level information, kept by the middleware, and reason about its own architecture in order to perform reconfigurations [2]. This middleware also facilitates implementing dynamic adaptation and software evolution activities.

The mapping of modules, ports and connectors to an implementation depends on the particular environment. In our prototype environment, primitive module types are defined by Java classes, and composite modules can be composed by arbitrary configurations of primitive modules (it is also possible to compose modules using Java's inheritance features, but this would imply in loosing the capability of reconfiguring the individual composing 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 configuration level; the remaining methods use normal Java referencing and binding mechanisms. Connectors 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.

4 Conclusions

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 of 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 component interface signature [1]. A prototype GUI to access the R-RIO's features was also developed. With the GUI, one can graphically design and run software architectures.

We are validating our research in two ways: (i) developing examples upon a prototype of the reflective middleware, and (ii) showing that CBabel descriptions are suitable to formal proofing of properties.

Currently, we are working on CBabel's QoS contracts. The idea is to have an open mechanism to allow the addition of QoS aspects and also define their implementation mapping on the R-RIO's configuration model [3]. Examples with QoS aspects are under development.

A preliminary version of R-RIO is available for research use; see http://www.ic.uff.br/~rrio for details.

Acknowledgments

This work has been partially supported by the following Brazilian research funding agencies CNPq, CAPES, Finep and Faperj.

References

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

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

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