On the Integration of Configuration and Meta-Level Programming Approaches

Orlando Loques[1], Alexandre Sztajnberg[2], Julius Leite[1], Marcelo Lobosco[1]

[1] Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brazil
{loques, julius,lobosco}@ic.uff.br
[2] Instituto de Matemática e Estatística/UERJ, RJ, Brazil
Grupo de Teleinformática e Automação/COPPE/UFRJ, RJ, Brazil
alexszt@uerj.br

Abstract. Configuration Programming, based on Architecture Description Languages, and Meta-Level Programming are considered promising approaches in the software engineering field. This paper shows that there is an immediate correspondence between some key concepts of Configuration and Meta-Level Programming approaches and that some of the main issues to be solved for their deployment in real systems are quite similar. The main result is that the integration of both approaches in a single configuration programming framework can assist in employing meta-level programming in order 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. A prototype of a configuration programming centered environment and some application examples are presented in order to demonstrate the useful features of the combined approach.

1 Introduction

Modern computer applications must be developed rapidly in order to meet increased market demands and fierce manufacturer competition. In many cases, different variants of a basic functional system have to be delivered in a short time, in order to cater to specific consumer requirements and particular operational environments.

Two extra requirements complete the current software engineering scene: (i) the components of these systems can be independently designed and may be implemented using different programming languages; (ii) many applications of interest have to change their component makeup during their operational life, i.e., they have dynamic architectures.

In the previously described context, software reuse becomes mandatory and compositional system development and separation of concerns are two key concepts that have been proposed to attain this goal. Another desirable characteristic is the capacity to support dynamic architectures and include components programmed in diverse languages in these architectures.

Architecture Description Languages (ADLs) are a convenient way of specifying a software architecture, either textually or graphically. This consists of the composition or configuration of the system in terms of a set of *modules* (components), which desirably encapsulate functional computation, and a set of *connectors*, which primarily describe how the modules are glued into and may interact within the architecture [1], [22], [31]. Using an ADL, a system designer can specify the functional composition of the system , through module selection, and attach to it particular module interaction styles or contracts (in communication contexts they are called protocols) through the connectors. For convenience, we call this activity Configuration Programming (CP).

Meta-Level Programming (MLP) in its different versions, such as computational reflection [6], [11], compositional filters [3] and aspect-oriented programming [13], [27], is being proposed as an approach that helps to achieve separation of concerns. MLP facilitates concentrating application concerns related to particular system requirements, that are in principle (hopefully) orthogonal to the pure functional behavior of the application, in explicit pieces of code, that together define the so-called meta-level architecture of the system. Reification, the switching process between the normal program (base-level) and the extra program (meta-level), can be implemented in several ways, depending on the particular MLP environment being considered. For example, in a procedural language context, specific procedure calls can be intercepted in order to transfer the control flow to the extra code. This is quite similar to the code control transfer that happens between a module and a connector in configuration programming contexts. This fact paves the way for integrating both paradigms in order to benefit from their intrinsic advantages.

Like other researchers, we are investigating techniques that may facilitate the combined use of the best of both concepts, CP and MLP, by system designers. This goal entails some simple insights and in the future may allow the construction of very powerful tools and integrated support environments to design, build, document, verify, operate and maintain software architectures. This happens mainly because Configuration Programming, based on ADLs, allows us to describe many static and dynamic system characteristics in very explicit forms that map naturally to the actual system software structure. This exposition makes it easier for the designer to understand the system architecture and to make the interventions required for customization, either at the application development stage or during its operational life, in order to cater to evolutionary requirements.

The remainder of this paper is organized as follows. Initially, in section 2, we present basic ADL/CP concepts and in the following section some of their implementation details in a specific environment. In section 4 a set of examples is used to illustrate our proposition. In sections 5 and 6 we discuss some related proposals and issues that are common in the context of both (MLP and CP) approaches, respectively. Finally, in Section 6, we draw some conclusions.

2 Basic Concepts

Software architecture is an emerging field still without a universally accepted terminology; in order to describe our proposition, we define some basic concepts on the ADL/CP area.

Module: A single or structured component with an independent existence, e.g., a process, object or procedure, a set of processes, objects or procedures, or any identifiable composition of code or data. A module functions as a type or class when used as a template to create execution units, and, as an execution unit it is called a module or class instance, or simply an object.

Port: A typed object that identifies a logical point of interaction of a module and its environment. The set of ports of a module defines its interface. Port type definitions can be reused in different modules to define their particular interfaces. A port instance is named and referenced in the context of its owner module (interface definition and code); this provides configuration-level port-naming independence. We distinguish ports that are active and can initiate interactions, called outports, from ports that are passive in the interaction, called inports; outports can be mapped to method calls and inports can be mapped to method entry points in an object code. Ports are similar to reification points used in MLP approaches and, similarly, can be associated with different code boundary transfer mechanisms, e.g., method, procedure or messaging passing primitive invocations.

Connector: an object that is used to relate ports or module interfaces. Besides this the connector encapsulates the contracts [12] used for module interaction. Connectors have properties similar to modules and can either be implemented by modules written in a particular programming language, or by run-time facilities of a particular operating system, or by any mix of resources available in the supporting environment. Protocols used in distributed systems and pipes used in Unix systems are immediate example of specialized connector implementations.

As mentioned in [31], the architecture of a software system defines that system in terms of modules and of interactions among these modules; these interactions are expressed and implemented by connectors. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between system requirements and the elements of the constructed system. In this context, we can look a non-functional requirement (and many other particular types of requirements) as an aspect or concern, and by an agreed self-discipline map its corresponding implementation into a specific connector type. In this way connectors can be seen as meta-objects in MLP based contexts. According to our proposal, Table 1 summarizes the correspondences between the MLP and ADL/CP abstractions.

With the help of ADL constructs, modules and connectors can be composed in order to define new aggregated components that can be reused in different application systems. For example, some system requirements can be described at higher levels of abstraction, and specified through composite connectors that encapsulate the meta-configurations required for the support of these requirements, e.g., fault-tolerance support connectors [21].

Table 1. Correspondence between ADL/CP and MLP abstractions

MLP abstraction		**ADL/CP abstraction**
base-level components	<->	modules
reification points	<->	ports
meta-level components	<->	connectors

The ADL itself may be extensible in order to allow the addition of new connectors and at the same time let the configuration programming environment get to know their semantic properties and configuration rules. This is required for formal reasoning and verification of architecture specifications, and, in addition, it may help to simplify the syntax required to use some connector types, e.g., connectors that implement communication protocols. Achieving ADL extension capability while retaining such properties is a non-trivial task [1].

3 R-RIO

In this section we describe some details of R-RIO (Reflective-Reconfigurable Interconnectable Objects), a distributed configuration programming environment, based on Java, that we are currently developing [19], [32].

3.1 Implementation

As described in the previous section, 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 losing the capability of reconfiguring the individual composite modules). Ports are associated with Java method 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. 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 with module and connector instances through R-RIO's ADL declarations. At configuration time, module and connector instances are created as Java objects. Table 2 summarizes the Java to R-RIO mappings.

R-RIO's connectors are context-reflective (Figure 1) 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

Table 2. R-RIO to Java mapping

R-RIO		Java
modules / module instances	->	classes / objects
connectors	->	Encapsulated in classes / objects or mapped to native support mechanisms
ports	->	method signatures / invocations

signature, including the exception-handling definitions associated with each particular method. This capability is achieved using information provided by the Java Structural Reflection Interface to adapt, at run-time, the particular connector type to the context of signatures defined by the port interfaces of the set of modules for which it is mediating interactions. Specifically, for each specified port-to-port connection, the connector automatically acquires the structural information required for the manipulation of the parameters of the associated method and provides the support machinery required for its invocation. This approach makes it unnecessary statically to generate and keep either stubs or connector executable code for each specific pair of connectable ports in a repository. We took advantage of this capability to provide generic connectors in order to support common communication styles (Java RMI, sockets, multicast, etc) used in distributed applications. This same technique, could 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 achieved through replicated servers [21]. An application of context-reflective connectors is presented in sub-section 4.4.

The structure of R-RIO's connectors follows a standard pattern and can be programmed according to a well-defined procedure, whose implementation could be supported by a tool. Basically, on the invoker side, after identifying the method, its invocation is serialized. In the sequel, on 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 using the standard Java virtual machine. Also, we have not put much effort in optimizing our connector implementation. However, we have made a set of measurements that show that the overhead is small when compared to the cost associated with the basic machinery required to perform dynamic method invocations in Java, which is considerably greater than the cost of normal method invocation. The connector's performance can be optimized either using precompiled stubs (losing the dynamic context-reflective capability) or using a customized JVM (losing portability).

3.2 Configuration Management

R-RIO includes a configuration management service that is used to create running images and to reconfigure already executing application architectures. The

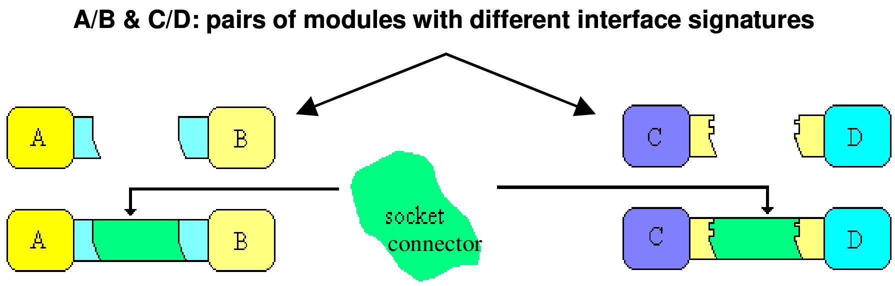


Fig. 1. Context-Reflective Connectors

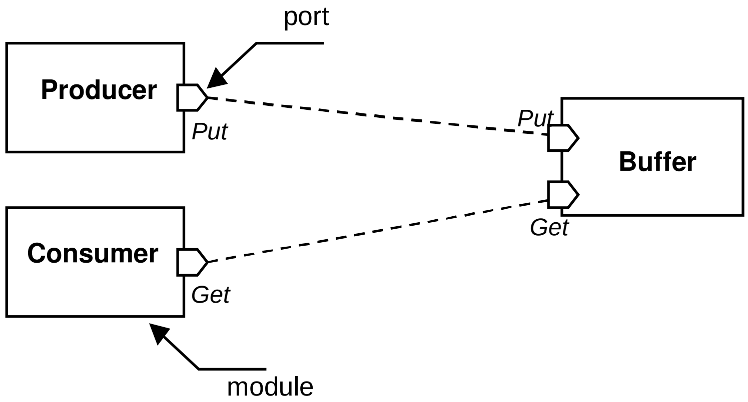
configuration management service has a layered structure that is implemented by two types of system components: (i) an executive, that provides basic methods for configuration control: module and connector instantiation/deletion as well as binding/unbinding of modules and connectors; and (ii) a manager, that provides higher level configuration management functions. The manager is based on an interpreter that takes architecture specifications (ADL scripts) and generates the basic executive method invocations to create a running image of the specified architecture. R-RIO also supports configuration changes in a running architecture (see sub-sections 4.3 and 6.2 for related points).

Currently, the configuration management service has a centralized implementation (i.e. there is only one manager component) and the basic configuration executive component is replicated in every processing node included in an R-RIO context. However, the configurator service could have a more distributed implementation, i.e., separated managers can be used to take care of different parts of a running architecture.

4 Examples

In its origins, our research had a distributed systems orientation. Most of the previously mentioned examples of connectors are well suited to the message communication area. However, we came to believe that connectors can be used to mediate general concerns related to module interactions. A similar view of connectors is presented in [31] where a scheduling connector is used to mediate real-time requirements of a set of modules. In our project, as exemplified in this section, we were able to express through connectors coordination (synchronization and concurrency) requirements similar to those expressed using the aspect-oriented language for coordination proposed in [20]. We also managed to encapsulate into connectors the machinery required to support fault-tolerance through server replication (see section 5.1), and found that diverse general purpose module interaction patterns can be mapped to connectors.

In this section we also outline additional points of our proposal. Initially, we customize a simple producer-consumer buffer application, that is also presented in other related works, e.g., [3], [20]. Finally, we show how the contextual-reflection capability of our connectors can be used for improving reuse. For clarity, in the examples, we avoided using configuration parameters, and ignore the error handling issue. CBabel, R-RIO's ADL, is described in [24] and [32].




```
module  BufferApplication {
  port Put (Any Item)(void);
  port Get (void)(Any Item);
  module   BufferType{
      inport Put;
      inport Get;
  } Buffer;
  module   ProducerType {
      outport Put;
  } Producer;
  module   ConsumerType {
      outport Get;
  } Consumer;
  install   Buffer, Producer, Consumer;
  link Producer, Consumer to Buffer;
} Example;
start Example;
```

Fig. 2. Producer-Consumer Buffer Architecture

4.1 Basic Buffer Architecture

The graphical and textual representations of this producer-consumer buffer architecture are presented in Figure 2. This basic version uses as connectors the standard Java method invocation support, that is the default in R-RIO; they are represented by dashed lines. The signature of the ports `Put` and `Get` are defined independently; they can be reused to define the interfaces of the application modules that compose the architecture. `BufferType`, `ProducerType` and `ConsumerType` are module type declarations; `Buffer`, `Producer` and `Consumer` declare respectively instances of these modules. `Inport` and `outport` designate the effective directions of the flow of data arguments through the ports; they also allow a port to be renamed, e.g., `outport Put Store`, when required to identify different ports for configuration naming purposes. Install declares the module instances that will be created in the running system.

A map directive is available to specify the module instance programming language and to designate a specific code implementation for it; here it is omitted and Java is considered the default language. The module's port linkage concludes the architecture specification. The final start declaration tells the configurator to create an instance called `Example` of the `BufferApplication` type.

4.2 Introducing Coordination

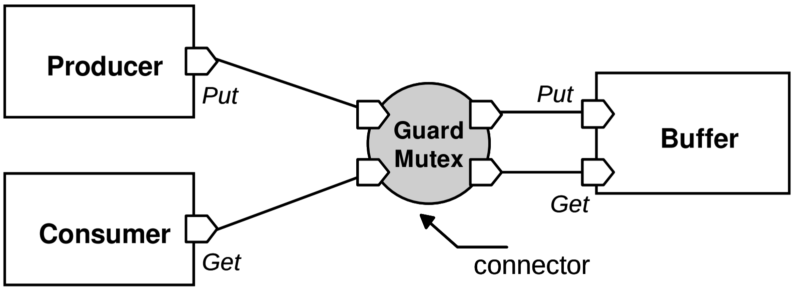
The basic buffer assumes standard Java semantics for method invocation and may not work well in the presence of concurrency even in the simplest case (one producer and one consumer). In Figure 3, we present the configuration description of a coordination connector that can be used to extend the buffer, adding mutual exclusion and flow control for the buffer access operations, allowing the inclusion and safe operation of multiple concurrent producers and consumers.

The configuration description uses standard declarations of R-RIO that may be used to implement similar requirements in other applications. `Exclusive` specifies that method invocations through enclosed ports (`Put` and `Get`) are mutually exclusive. Guards (`empty` and `full`) are used to control the flow of method invocations; if the guard is false the associated method invocation is deferred. The state variable `n_items` has to be acquired in the buffer data space. This was done through a method coded by hand; however it could also be automatically generated. One instance of this connector can be included in the configuration using primitive configuration language constructs. However, when the interfaces of the connectors and of the involved modules are paired and context compatible a simplified notation may be used:

```
link  Producer, Consumer to Buffer by Guard_Mutex;
```

4.3 Introducing Distribution and Encryption/Decryption

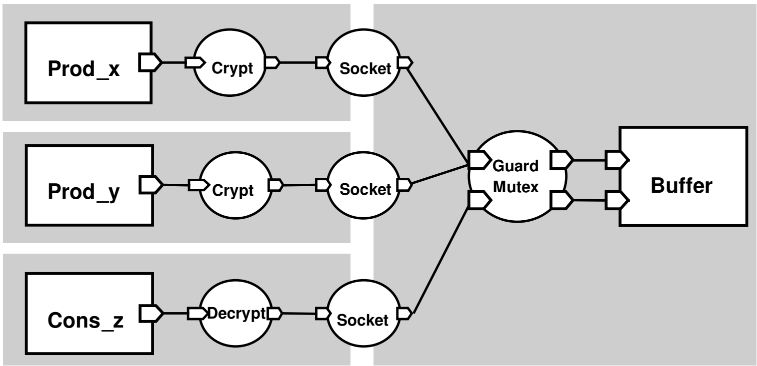
Here a distributed environment (producer, consumer and buffer are in different nodes) is considered. The distributed architecture and part of its configuration




```
connector Guard_Mutex{
    condition empty = true, full = false;
    state_required int n_items;
    int MAX_ITEMS = 10;
    inport Get; inport Put;
    exclusive {
      outport Put {
        guard  (empty) {
          after {
            full = true;
            if (n_items == MAX_ITENS)
              empty = false;
      } } }
        outport Get {
        guard  (full) {
          after {
            empty = true;
            if (n_items == 0)
              full = false;
      } } }
    }
}
```

Fig. 3. Producer-Consumer Buffer Architecture with a Coordination Connector

specification in R-RIO are represented in Figure 4. An `at` directive is available to specify the module's location (this could be parameterized); the shaded areas represent different nodes. As a default communication protocol R-RIO offers a Java-RMI connector; in this example we chose to use a connector that directly implements socket communication. Note that socket is a composite connector, with a distributed implementation. In principle, at the application code level, the introduction of the module distribution aspect is invisible; see the related discussion in section 5.4. At this stage we could also add encryption/decryption support to the architecture.




```
...
module     ProducerType Prod_x, Prod_y;
module     ConsumerType Cons_z;
install    Prod_x, Prod_y, Cons_z;
             at node_i, node_j, node_k;
link       Prod_x, Prod_y, Cons_z to Buffer
             by Crypt|Crypt|Decrypt, Socket, Guard_Mutex;
start      Prod_x, Prod_y, Cons_z;
...
```

Fig. 4. Producer-Consumer Buffer Architecture with Encryption/Decryption and Socket Connectors

In general, the capability to perform dynamic reconfiguration requires special system or application programming in order to ensure overall application state consistency [15]. For example, in the previously presented architectures, the removal of producer or consumer instances could lead the application to an inconsistent state, e.g., an item value could be lost. Also, in the architecture presented in figure 4, the initial introduction of encryption/decryption would require the buffer to be empty and has to be in force from system start-up. However, during operation, an arbitrary number of producer or consumer instances can be introduced through dynamic reconfiguration of the running architecture without causing any buffer state inconsistency. The following configuration script can be used to introduce a new consumer:

```
module   ConsumerType Cons_t;
install  Cons_t at node_l;
link     Cons_t to Buffer
          by Decrypt, Socket, Guard_Mutex;
start    Cons_t;
```

We further discuss dynamic reconfiguration in sub-section 6.2.

4.4 Using Context-Reflective Connectors

It is interesting to note that in the buffer architecture versions, presented in the previous sub-sections, it was unnecessary to define the data type of the buffered items. In fact, the only requirement for the presented configurations to work correctly is that the data type of the `Item` parameter, associated with the Consumer's and the Producer's `Put` and `Get` ports, be identical. This happens because the contextual-reflection capability provided by the deployed connectors. As described in section 3.1, guided by the configuration specification, the connectors can acquire from the Java environment the structural information needed to manipulate and invoke the method associated with each port-to-port binding.

Currently, we 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 [4]. Besides increasing design reuse, this allows us to automate code generation for supporting the specified pattern in different application architecture contexts. Figure 5 helps to illustrate the context-reflective capability: the standard observer design pattern, as defined in [10], is implemented through a generic connector used to mediate the interactions of a Tic-Tac-Toe game architecture. In this example, after each move, the connector gets the new game status (invoking the game method `GetStatus`) and forwards it to the display modules, using a multicast connector. In fact, this basic architecture can be adapted automatically to support several other board games having similar architectures. For this adaptation it would be sufficient to replace the application specific classes: `TTTplayer`, `TTTgame` and `TTTdisplay`.

A kind of context-reflective connector is also proposed in [26] in order to support architectural evolution. However, these connectors are designed to bind components using a specific architectural style (called C2), which imposes a particular structure for the application system. The connector's main responsibilities are the routing and broadcast of messages; the messages can be filtered internally by the connector, according to some specific policies. In addition, the proposed implementation does not support dynamic configuration changes.

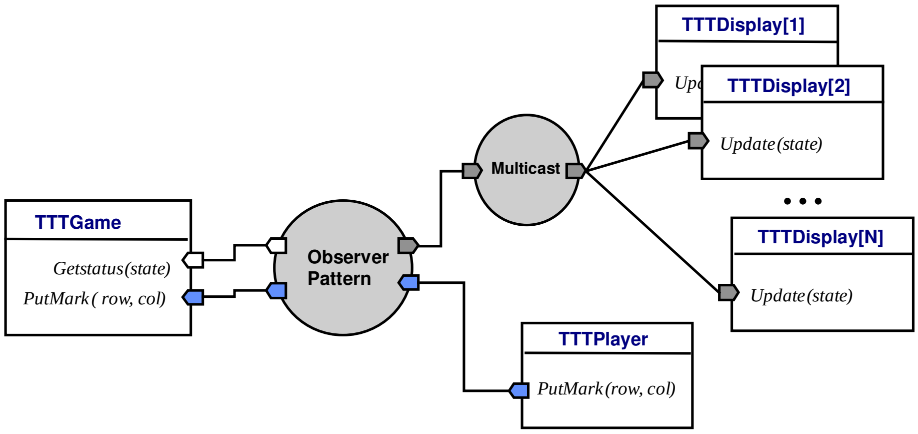


Fig. 5. A Context-Reflective Connector

5 Related Proposals

In this section, we discuss two approaches directly related to our proposal. A more extensive discussion is available in [32].

5.1 Reflection

Reflection is a paradigm, generally associated with object-oriented languages, with great potential to help to achieve separation of concerns. In practice, method interception (and redirection) is the primitive mechanism available to allow the use of reflection by programmers. By intercepting a method invocation, defined to have a reflex, the programmer can introduce some additional code (defined at a meta-level and sometimes called meta-code or meta-object) to introduce or add specific properties to the program. In fact, in many cases, this is equivalent to introducing a connector to glue together two or more module ports; the specific connector is specified in the configuration programming domain. For example, in a client/server application connectors can be used to fulfill requirements such as: call logging and tracing, persistence, data compression and encryption, access control, real-time and fault-tolerance.

One of our initial research goals was to compare the configuration programming and computational reflection approaches for application system customization. To begin with, we used the reflection technique to implement fault tolerance support in CORBA using the set of server replication policies presented in [7], which was originally implemented in Open C++ [6]. As a result, we obtained a set of meta-objects that encapsulate the fault tolerance support mechanisms for the chosen server replication policies [8]. Then, we verified that the fault tolerance code developed for the meta-objects can be used, without basic changes, to implement a generic connector that supports the same set of server replication policies presented in [7]. Using this special connector we were able automatically to add replication-based fault tolerance policies to CORBA-based client-server architectures [21]. This fault tolerance experiment showed that, at the programming level, configuration requires about the same effort as computational reflection and, as a benefit, the implementation is independent of special programming languages and compilers. We were also able to reproduce, through configuration and connectors, other programming examples presented in the literature to illustrate the use of reflection-based and other related proposals, e.g., [3], [11].

5.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a new MLP paradigm being proposed to cater for separation of concerns [20] and [27], which in this context are considered as aspects of an application. According to the first AOP proposal, the implementation of a system in a particular programming language consists of: (i.i) a component language with which to program the components, (i.ii) one or more aspect programming languages with which to program the aspects or concerns, (ii) an aspect weaver for the set of languages, (iii.i) a component program, that implements the (functional) components using the component language, and (iii.ii) one or more aspect programs that implement the aspects using the aspect languages.

The weaver acts as a special compiler that combines the functional and aspect codes to implement a single running program. Analyzing the AOP literature, two types of code transformation provided by the weavers can be identified. The first, that does not appear to happen very frequently, consists of optimizations, guided by special directives specified at the aspect program level, that lead to more performance-efficient programs. The authors argue that, in this class of applications, the program analysis and understanding involved is so significant that compilers cannot be relied upon to do the required optimizations reliably; this would happen because the (meta and base) code join points cannot be treated separately [18]. We think that such kinds of optimizations are in most cases orthogonal to the overall system architecture. They can be dealt, at module or subsystem level, using special tools, without hindering the overall system composition. In ADL/CP terms, the aspect language program, its weaver and also the run-time environment support could be attached to specific modules or subsystems through related annotations, made at the configuration programming level. In [22] we suggest the use of such kind of annotation to select the data coherency protocol to be used to support module interactions in a distributed-shared memory (DSM) context.

The second code transformation style, which appears to be more common, is used when the code join points can be treated separately, and in essence is similar to that performed by reflective language compilers. In fact, as pointed out in [27] "reflection seems to be a sufficient mechanism for AOP, but many regard it as too powerful: anything can be done with it, including AOP". In this context, an advantage of AOP would be its ability to discipline the use of reflection by using special aspect programming languages. The Java based language, AspectJ, is a step in this direction, providing a construct: advise, that is used to specify the methods that are to be logically intercepted and the associated meta-level code to be executed [13]. In fact, the *advise* role is similar to that performed by bind constructs in ADL/CP based proposals, and this can be a bridge between the two approaches.

In our project, we were able to reproduce an application of the observer pattern presented in [13], coded in AspectJ, using connectors to specify and concentrate the code required to implement the mentioned pattern [32]. Regardless of the case with AOP, we did not have to use any special aspect programming language, besides some constructs that are almost standardized in most contemporary ADL/CP proposals. Yet, as regards the implementation of the observer pattern in AspectJ, we do not have to break module encapsulation in order to introduce special methods, method calls and data structures into the basic application modules. It should be pointed out that our solution assumes that the observed module provides a method to expose its internal state. However, this programming style requirement is very reasonable in context of the specific application.

6 Related Issues

In this section we discuss issues that are common to the MLP and ADL/CP approaches and try to relate these issues to our proposal.

6.1 Composition of Concerns

As pointed by other researchers, composition of concerns/aspects is an open research issue; e.g. [28], [35]. We have to find appropriate syntactic mechanisms to structure the meta-level architecture and also find well-defined semantic rules to transfer control and data among meta-level components. In addition, techniques to reason about the composite meta-level architecture have yet to be developed. In this area, the ADL/CP approach already provides clear concepts and mechanisms for module and connector composition. In particular, we can benefit and take advantage of the techniques and formalisms being proposed to reason and verify correctness properties of software architectures, e.g., [1], [2], [23].

In our proposal meta-level code is associated with connectors, that can be composite components and thus can have any convenient internal structure. In order to improve flexibility for meta-level programming, R-RIO provides a special *transfer* programming construct, which is designed to pass control and data through composite connector structures. The implementation of this construct propagates the reference of the base-level module, which called the method that started the reification process, among the connectors through which that method execution traverses. This allows control to come back to the original module after traversing an arbitrary configuration of meta-level objects. The use of transfer also caters for efficiency in some configurations of connectors (e.g. pipelines), allowing us to bypass all (or some) of them, on the way back, before returning control to the invoking module. Extra fine performance optimizations are possible using configuration analysis techniques to unite together the code for individual connectors into a single piece of code, thus avoiding overheads related to method invocation, data transfer, context changes, etc.

Another issue for composite connectors is how the components of a composite meta-level architecture gain control over a base-level module, for example to access and modify its state. When there is no internal concurrency, either at the meta or base level, there are solutions that are orthogonal to specific run-time support environments. For example, in a state checkpoint application, special methods (supplied at the base level) can be used to get and update the application state to be preserved. However, in the presence of concurrency at either (base or meta) level, synchronization and race conditions appear and more refined mechanisms may be required. In some cases it is possible to identify *a priori* particular interaction styles between the two levels, e.g., as happens in a distributed transaction support framework. General purpose solutions are presented [11] and [28]; however they either restrict the meta-base level interaction relationship or rely on special features, which impose, changes in the basic support environment (Java virtual machine), hindering portability.

6.2 Dynamic Architectures

The architecture of some applications may have to be modified or extended during operation in order to cater for either planned or evolutionary (or unplanned) change. In this regard, the ADL/CP approach provides the capability of describing and managing system composition and structure explicitly and independently of application programming concerns, thus helping to attain separation of concerns in the configuration programming domain. On the other hand, most MLP based approaches do not either allow or offer clear concepts and mechanisms for supporting dynamic adaptation leading to ad-hoc solutions in this domain of concern.

In our proposal, for planned reconfigurations, we achieve explicit separation of reconfiguration actions through a special 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 configurator method. Other researchers propose extended ADLs with the capability of expressing configuration management policies explicitly, e.g. [5], [16]. In [5] a scheme, called architectural reflection, is presented, in which reconfiguration activities have both their decision-making policy and the module/connector structure required for changes specified separately at a meta-configuration level, which is defined by special ADL constructs. Reconfiguration activities are triggered by the activation of rules (also specified through special ADL constructs) based on state and events related to the base level architecture computations. Although state and event-based internal reconfiguration triggering rules could have been added to CBabel (R-RIO's ADL), we opted for keeping our ADL concise, introducing decision-making concerns explicitly where necessary.

In fact, R-RIO supports a pragmatic version of the concepts proposed in [5] and could be used as a basis for a more canonical implementation of those concepts. Our approach conforms to the idea of providing skeleton ADLs to which to add the particular details of concern when necessary [17]. For instance, in a fault-tolerance experiment, based on server replication, we used a group communication service to implement special connectors in order to support diverse fault-tolerance styles [21]. The used group communication service already provides most of the information and mechanisms required for triggering and executing consistent reconfiguration activities [33]. Hence, we opted for implementing the reconfiguration decision-making code as a meta-component of the fault-tolerance support composite connector. For recovery after failures, the configuration changes, explicitly specified by a separate script at the ADL level, are triggered by invocations made from the fault-tolerance connectors. In this way, we separate both concerns and leave the configuration management functions explicitly encapsulated into a component.

R-RIO also supports unplanned configuration changes that can be performed through an interactive configuration interface, which is provided by the configuration management service. The change commands invoked through the configuration interface are reflected in the internal representation of the currently running architecture. After a sequence of configuration change commands, the resulting running configuration can be automatically converted to the corresponding textual specification in CBabel. It is feasible to analyze and validate an intended final architecture before imposing the actual changes in the current configuration. We have also experimented with a graphical interface that allows managing the evolution of software configurations by graphical means [22].

Other schemes for adaptive programming, such as that used in the Darts system [29], offer a limited set of adaptive policies, and the implementation mechanism for switching among these policies is embedded in the support environment. The programmer is restricted to adapting the running software configuration through directives called from application modules during system operation. Configuration programming provides flexibility and does not forbid the use of dynamic configuration support provided by current operating systems projects, such as Darts.

It is interesting to note that code fusion techniques, like those implemented by AOP weavers, can mix the basic functional and specific (meta) code pieces together, in order to compose the running application code. As a result, the run-time granularity of the reconfigurable code units is at least increased, which makes it difficult to support the dynamic evolution requirements of some applications; this was also pointed out in [25].

6.3 Implementation Flexibility

Our ADL includes an Interface Definition Language (IDL), used to specify the component's interfaces, and has a syntax compatible with CORBA IDL. In principle, the adoption of a standard IDL would help us to use in an application system architecture components implemented in different programming languages, provided they have in common a CORBA support environment. We have carried out independent configuration programming experiments using Java and CORBA support environments [19] and [21]. The Java implementation provides interoperability and component portability in standard Java environments. While not fully developed, the CORBA implementation promises to support interoperability for components written in different programming languages. It is feasible to construct automatically connector types which serve as bridges between these two environments (as well as between other support environments like Microsoft's DCOM); the information required is readily available using the Java Structural Reflection Interface and CORBA interface repositories. These adaptation bridges could also be obtained using conversion libraries provided by CORBA and Java products. The general availability of these bridge connectors would allow interoperability among modules being supported in different environments. Like others researchers we think that interoperability is essential to support current software engineering requirements [34].

6.4 Comments on Reuse

ADL/CP and MLP approaches provide clear concepts and powerful tools to compose and deal with evolving software architectures. However, either in ADL/CP or MLP environments, the immediate reuse of independently designed modules and connectors (or meta-objects) may not be possible. In some cases, in order to choose or design a connector for a given module application context, the designer has to know how the concerned modules interact and even some of the internal details of these modules.

Two examples illustrate this point: (i) in the architecture presented in subsection 4.4 the game modules have to follow a particular design style to take advantage of the observer pattern connector mediating capability; (ii) connectors, used for communication in distributed systems, are generally considered "almost" orthogonal to the applications. However, when we look at the exception handling issue things get blurred. The mechanical side of exception propagation and handling can be standardized and automated. However, the complete treatment of an exception may require the intervention of the application programmer through specific coding, sometimes disseminated into several modules in the architecture. This has to be planned when designing the application, and requires appropriate module internal and external architectures. To summarize, there are limits on what can be solved independently at the meta-level. Perhaps, as observed by Saltzer, the treatment of end-to-end issues is inherently an application dependent concern [30]. Much more research is necessary to augment reuse in these cases.

7 Conclusions

This paper has shown that there is an immediate correspondence between some key concepts of Configuration and Meta-Level Programming. Its main result is that the integration of both approaches in a single configuration programming framework can assist in putting into practice meta-level programming in order to achieve separation of concerns and improving software reuse. Meta-level components can be mapped into connectors and application customization can be achieved by specifying these 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 particular, the availability of context-reflective connectors may enable the synthesis of highly dynamic architectures as those made possible in the WWW environment.

The ADL description of a system is itself a specification and one can take advantage of formalisms already available for refining and proving properties of software architectures. With relatively little effort, a concise ADL can be used as a basis to express different concerns best suited for developing applications in a given domain. For instance, starting from our basic ADL, we are investigating how to attach to components descriptions of QoS (Quality of Service) specifications (behavioral contracts), similar as those proposed in [9]. This allows two new interesting and complementary capabilities: (i) verify beforehand if functionally compatible modules will obey QoS interaction contracts in a distributed system architecture; (ii) to assemble composite connectors, which may include the configuration of specific support components, required to meet the specified QoS contracts, as suggested in [14]. The latter capability may help to resolve most of the resource dependencies of an application in an automatic fashion, with little or no external human intervention.

Currently, we are developing R-RIO, which integrates ADL/CP and MLP approaches in a single, conceptual and practical, configuration programming environment. This experiment should help us to clarify many issues that this work does not directly address. 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 Brazilian research funding agencies CNPq, Finep and Faperj. Alexandre Sztajnberg is supported in part by a grant from CAPES - Brazil, proc. BEX 1014/99-4. We would like to thank our colleague Michael Stanton for his valuable suggestions for the improvement of the text, and also the anonymous referees for their constructive remarks.

References

1. Allen, R., Garlan, D., A Formal Basis for Architectural Connection, in ACM Transactions on Software Engineering and Methodology, July 1997.

2. Astley M., Agha G.A., Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management, in proc. Sixth International Symposium on the Foundations of Software Engineering, ACM Software Engineering Notes, Vol. 23, N. 6, pp. 1-9, 1998.
3. Bergmans L.M.J., Composing Concurrent Objects - Applying Composition Filters for the Development And Reuse of Concurrent Object-Oriented Programming, Ph.D. Thesis, Universiteit Twente, Holland, 1996.
4. Carvalho, S. T., Integrating Design Patterns in Software Architectures, (in Portuguese), Research Report, IC/UFF, Brazil, 1999.
5. Cazzola, W., Savigni, A., Sosio, A., and Tisato, F., Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, DSI, University degli Studi di Milano, May 1999.
6. Chiba S., A Meta-Object Protocol for C++, in Proceedings of OOPSLA'95, ACM SIGPLAN Notices, Vol. 30, N. 10, pp. 285-299, 1995.
7. Fabre , J., et al., Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming, in proc. 25th IEEE International Symposium on Fault-Tolerant Computing, 1995.
8. Fraga, J., Maziero, C., Leung, L.C, Loques, O.G., Implementing Replicated Services in Open Systems Using a Reflective Approach, The Third International Symposium on Autonomous Decentralized Systems, Berlin, Germany, pp. 273-280, 1997.
9. Frolund, S.; Koistinen, J.; Quality-of-service specification in distributed object systems, Distributed Systems Engineering Journal, V. 5, pp. 179-202, 1998.
10. Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object Oriented Software, Addison Wesley, 1995.
11. Golm, M.; Design and Implementation of a Meta Architecture for Java, M.Sc. Thesis, Erlangen-Nremberg University, Germany, 1997.
12. Helm, R., Holland, I.M. e Gangopadhyay, D., Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, OOPSLA'90, pp. 303-311, 1990.
13. Kiczales G., Lopes C., Aspect-Oriented Programming with AspectJ, Xerox PARC, www.parc.xerox.com/aop, 1998.
14. Kon F., Campbell R. H., Supporting Automatic Configuration of Component-Based Distributed Systems, in Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, San Diego, 1999.
15. Kramer J., Magee J., The Evolving Philosophers Problem: Dynamic Change Management, IEEE Transactions on Software Engineering, V. 16, No. 11, pp. 1293-1306, 1991.
16. Kramer, J. and Magee, J., Dynamic Structure in Software Architectures, Proc. of the Fourth ACM Sigsoft Symp. On Foundations of Software Engineering, California, USA, October 1996.
17. Kramer, J. and Magee, J., Exposing the Skeleton in the Coordination Closet, Coordination 97, Berlin, 1997.
18. Lamping J., The Interaction of Components and Aspects, ECOOP'97 AOP Workshop document, www.cs.utwente.nl/aop-ecoop97, 1997.
19. Lobosco, M. R-RIO: A Java Environment for Supporting Evolving Distributed Systems (in Portuguese), M.Sc. Dissertation, IC/UFF, Brazil, 1999.
20. Lopes, C.I.V.; D: A Language Framework for Distributed Programming, Ph.D. Thesis, College of Computer Science, Northeastern University, Boston, USA 1997.
21. Loques O., Botafogo R., Leite J., A Configuration Approach for Distributed Object-Oriented System Customization, Proceedings of the Third International IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Newport Beach, USA, pp. 185-189, 1997.

22. Loques O., Leite J., Carrera E.V., P-RIO: A Modular Parallel-Programming Environment, IEEE Concurrency, Vol. 6, N. 1, pp. 47-56, 1998.
23. Luckhan D.C., et al., Specification and Analysis of System Architecture Using Rapide, IEEE Trans. on Software Engineering, Vol. 21, No. 4, pp. 336-355, 1995.
24. Malucelli, V. V, Babel - Building Applications by Evolution, M.Sc Dissertation (in Portuguese), DEE / PUC- RJ, Rio de Janeiro, Brazil, 1996.
25. Matthijs F., et al., Aspects should not die, ECOOP'97 AOP Workshop document, www.cs.utwente.nl/aop-ecoop97, 1997.
26. Medvidovic N., Architecture-Based Specification-Time Software Evolution, Ph.D. Dissertation, University of California, Irvine, 1999.
27. Mens K, Lopes C.V., Tekinerdogan B, Kiczales G., ECOOP'97 Aspect-Oriented Programming Workshop Report, www.cs.utwente.nl/aop-ecoop97, 1997.
28. Oliva A., Buzato L., Composition of Meta-Objects in Guaran, in Proceedings of the Workshop on Reflective Programming in C++ and Java, pp 86-90, OOPSLA 98, Vancouver, Canada, 1998.
29. Raverdy P-G., Van Gong H., Lea R., DART: A Reflective Middleware for Adaptive Applications, in Proceedings of the Workshop on Reflective Programming in C++ and Java, pp. 37-45, OOPSLA 98, Vancouver, Canada, 1998.
30. Saltzer J.H., et al., End-toEnd Arguments in System Design, in Proceedings of 2nd Int'l Conference on Distributed Computing Systems, Paris, France, pp. 509- 512, 1981.
31. Shaw M., et al., Abstractions for Software Architecture and Tools to Support Them, IEEE Trans. on Software Engineering, Vol. 21, No. 4, April 1995, pp. 314-335.
32. Sztajnberg, A., Flexibility and Separation of Concerns in Distributed Systems, Ph.D. Thesis proposal (in Portuguese), COPPE/UFRJ, Rio de Janeiro, Brazil, 1999.
33. van Renesse, R., Birman, K.P., Maffeis, S., Horus: A Flexible Group Communication System, Communicatons of the ACM 39, 4, pp 76-83, 1996.
34. Wegner P., Interoperability, ACM Computing Surveys, Vol. 28, No. 1, pp.285-287, 1996.
35. Welch I., Stroud, R., Dalang - A Reflective Java Extension. in Proceedings of the Workshop on Reflective Programming in C++ and Java, pp. 11-15, OOPSLA 98, Vancouver, 1998.