Towards Integrating Meta-Level Programming and Configuration Programming

Orlando Loques, Julius Leite, Marcelo Lobosco
CAA/UFF
{loques, julius,lobosco@caa.uff.br}

Alexandre Sztajnberg
IME/UERJ - GTA/COPPE/UFRJ
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 Programming and Meta-Level Programming and that some of the main issues to be solved for their deployment in real systems are quite similar. The main proposition 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 improve software reuse.

1. Introduction

Nowadays sophisticated computer application systems must be developed rapidly in order to meet increased market demands and great manufacturer competition. In many cases, different variants of a basic functional system have to be delivered in a short time, in order to cater for specific consumer constraints and particular operational environments, without incurring high costs. In this 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.

Architecture Description Languages (ADLs) are a convenient way of specifying, textually or graphically, software architectures: a system composition or configuration 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 with the architecture [2,20,25,29]. Using an ADL, a system designer can specify the system functional composition, 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 (M-LP) in its different versions, such as computational reflection [5,9], compositional filters [1,4] and aspect oriented programming [13,18], is being proposed as an approach that helps to achieve separation of concerns. M-LP facilitates concentration of concerns related to particular system requirements, and in principle (hopefully) orthogonal to the pure functional behavior of the system, 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 M-LP 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 transference that happens between a module and a connector in configuration programming

contexts. This fact paves the way for integrating both paradigms in order to take advantage of their intrinsic advantages.

Like other researchers, we are investigating techniques that may facilitate the combination by system designers of the best of both concepts: CP and M-LP. This goal entails some quite 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, allow 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 structure and make the interventions required for customization, that can be done using M-LP techniques.

The remainder of this paper is organized as follows. In the next section we present basic concepts and some of their implementation details in a specific environment. In section 3 an example is used to illustrate our proposition. In Section 4 we discuss some related proposals and issues that are common in the context of both (M-LP and CP) approaches. Finally, in Section 5, we provide some conclusions.

2. Basic Concepts

This section presents some ADL and configuration programming concepts, and also describes some details of their implementation in R-RIO (Reflective-Reconfigurable Interconnectable Objects), a configuration programming environment, based on Java, that we are currently developing [17,30,31]. Software architecture is an emerging field yet without an universally-accepted terminology definition; to convey our proposition, we use basic concepts on the ADL/CP area.

Module: A component with an independent existence, e.g., a process, object, procedure or any identifiable piece of code or data. A module functions as a type or class if used as a template to create execution units, and, as an execution unit is called a module or class instance.

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 M-LP approaches and, similarly, can be associated with different code boundary transfer mechanisms, e.g., method, procedure or messaging passing primitive invocations.

Connector: a typed object that is used to relate ports or module interfaces. Besides this the connector encapsulates the roles or contracts [2,10] 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 connector implementations.

As mentioned in [29], the architecture of a software system defines that system in terms of modules and of interactions among these modules; these interactions are expressed and

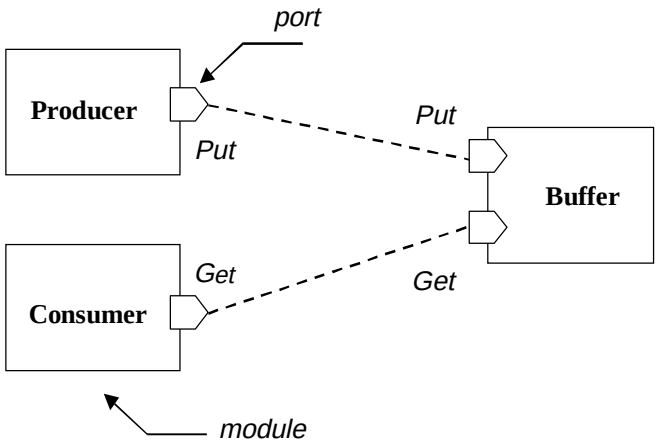
implemented by connectors. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between system requirements and the elements of the constructed system. In this context, we can look the non-functional requirements as aspects or concerns, and by an agreed self-discipline map their implementations into specific connector types. In this way connectors are similar to meta-objects or aspects.

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 applications. The mapping of modules and connectors to an implementation depends on the particular environment. In our prototype Java-based environment [17], module types can be defined by classes, ports can be associated to Java methods (that may be renamed at the configuration level in order to ensure configuration naming independence) and the connectors are also encapsulated in Java classes. It is important to note that only the methods explicitly associated with ports are configurable through connectors and directly visible at configuration level; the other methods use normal Java referencing and binding mechanisms.

The ADL itself should 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, configuration rules, and implementation requirements. Besides simplifying the immediate use of connectors, this is required for formal reasoning and configuration verification activities, and to simplify the syntax required to use some connector types, e.g., connectors that implement communication protocols. In addition, we want to facilitate the description of some system requirements at higher level of abstractions, using annotations to specify the meta-configurations required for the implementation of these requirements, e.g, fault-tolerance (see [19], for an example). Achieving ADL extension capability to automate the support of such kind of connectors is a non-trivial task [2,8,30].

Currently, R-RIO provides a basic set of connector types that can be extended as required. These connectors are unambiguously known by the supporting tools and can be automatically specified by configuration. Generic connectors to support different communication styles (Java RMI, sockets, multicast, etc) are available. When plugged into modules, these connectors are automatically adapted to mediate interactions between ports using any method signature, including exception definitions. This same technique can be used to make generic connectors to encapsulate functions that act on well-defined data structures, e.g., encryption/decryption, compression/decompression, state saving and fault-tolerance [19]. We are also able to express through connectors and configuration language annotations coordination requirements (synchronization and concurrency) similar to those expressed using the aspect-oriented language for coordination proposed in [18]; an example is presented in next section.

The programmer can also write new connectors in Java to encapsulate any aspect or concern associated with module interaction. These connectors are treated as normal modules and are specified using the configuration language constructs.




```
module  BufferApplication {
  port Put (int Item)(int Ack);
  port Get (void)(int 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.1 Producer-Consumer Buffer Architecture

3. An Example

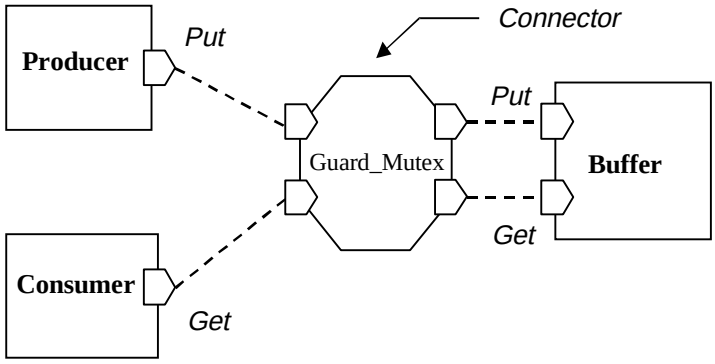
In this section we outline through an example some points of our proposition; we use a buffer application that is also presented in other related works, e.g., [4, 18]. For clarity, we avoided using abbreviated syntax constructs, configuration parameters, and ignore the thorny error handling issue. CBabel, R-RIO's ADL is described in [22,30].

3.1 Basic Architecture

The graphical representation of this configuration is presented in figure 1. 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 convenient 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 modules port linkage concludes the architecture specification. The final `start` declaration tells the configurator to create an instance called `Example` of the `BufferApplication` type.

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

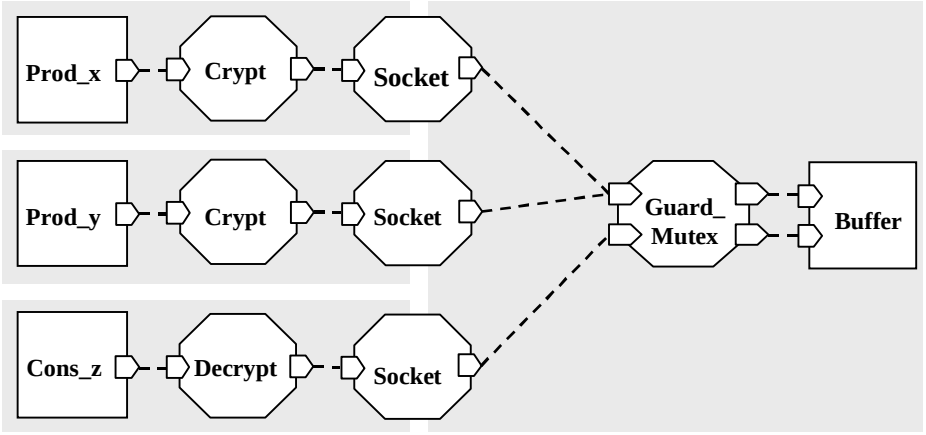



```
connector Guard_Mutex{
    condition empty = true, full = false;
    state_required int n_items;
    int MAX_ITENS = 10;
    inport Get;
    inport Put;
    exclusive {
      outport Put {
        guard  (empty) {
          after {
            if (n_items == MAX_ITENS) {
             empty = false;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

Fig.2 Architecture with a Coordination Connector

3.2. Introducing Coordination

The basic buffer assumes standard Java semantics for method invocation and may not work well in some situations, e.g., multiple concurrent producers and consumers. In figure 2, we present the configuration description of a cordination connector that can be used to extend the buffer, adding mutual exclusion and flow control for the buffer access operations. 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 directly compatible a simplified notation may be used:




```
...
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.3 Architecture with Encryption/Decryption and Socket Connectors

3.3. Introducing Distribution and Encryption/Decryption

Here a distributed environment (producer, consumer and buffer are in different nodes) is considered. An `at` directive is available to specify the module distribution aspect (that 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 choose to use a connector that directly implements socket communication. Note that socket is a composite connector,

with a distributed implementation. In principle, at the code level, the introduction of the distribution aspect is invisible; see the related discussion in section 4.6. At this stage we could also introduce encryption/decryption support into the architecture.

Additionally, an arbitrary number of producer and consumer instances can be introduced in this buffer architecture (this could be done during its operational stage). These instances can be specified giving different names for them or using indices to describe, install and link arrays of components. The final architecture, and part of its configuration specification in R-RIO, is represented in figure 3.

4. Related Proposals and Issues

In this section, we present from a high level point of view some related works and discuss some issues related to our proposition. In order to simplify the presentation, the discussion implicitly assumes an object-based context, but in principle the results presented may also be applied to class-based contexts, which would allow us to take full advantage of object-oriented technology properties.

4.1. 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 goals was to compare the configuration programming and computational reflection approaches for system customization. In a first stage, we used the reflection technique to implement in CORBA the set of module replication policies presented in [6], which was implemented in Open C++ [5]. As a result, we obtained a set of meta-objects that encapsulate the fault tolerance support mechanisms for the chosen fault tolerance policies [7]. 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 replication policies [19]. Using this special connector we can automatically add replication-based fault tolerance policies to any CORBA-based client-server architecture. This fault tolerance experiment has shown that, at the programming level, configuration requires similar efforts to computational reflection and, as a benefit, the implementation is independent of special programming languages and compilers.

We were also able to reproduce without great effort, through configuration and connectors, other programming examples presented in the literature to illustrate the use of reflection-based proposals, e.g., [4,9].

4.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a new M-LP paradigm being proposed to cater for separation of concerns [11,13,18,24], that in this context are considered as aspects. According to the first AOP proposal, the implementation of a system in a particular programming language consists of: (i.a) a component language with which to program the components, (1.b) one or more aspect programming languages with which to program the aspects or concerns, (ii) an aspect weaver for the set of languages, (iii.a) a component program, that implements the (functional) components using the component language, and (iii.b) 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, e.g., [11,16], 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 [16]. 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 [20] we suggest to use 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, that 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 [24] "reflection seems to be a sufficient mechanism for AOP, but many regard it as too powerful: anything can be done with it, including AOP. When using reflection, will aspect-oriented programming be safe or efficient enough ? Is reflection required to make the program adaptable enough ?" It was also argued that what was missing to constrain the power of reflection is a composition methodology at the meta-level. In this context, an advantage of AOP would be its ability to discipline the use of reflection through the use of 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 very similar to that performed by link or bind constructs in ADL/CP based proposals, and this can be a bridge between the two approaches. In our proposal, when convenient, a primitive like advise could be used to introduce application defined code into connectors [30].

In our project, we were able to reproduce the examples presented in [13], coded in AspectJ, using connectors to specify and concentrate the code required to implement the aspects [31]. 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 to 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.

4.3. Composition

As pointed by many researchers, composition of concerns/aspects (or connectors) is an open issue; e.g. [26,33]. 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 the 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., [2,3,21,25]

In order to improve flexibility for meta-level programming, R-RIO provides a special transfer construct, that is designed to pass control and data through composite connector structures. The implementation of this construct propagates the reference of the base-level module, that 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 (connectors). 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 compact the individual connector codes into a single piece of code, thus avoiding overheads related to method invocation, data transfer, context changes, etc.

How the components of a composite meta-level architecture gain control over a base-level module, for example to access and modify its state, is an issue also for composite connectors. 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, special methods can be used to get and update the required state. However, in the presence of concurrency at either (base or meta) level, synchronization and race conditions appear and more refined mechanisms may be required. We are investigating high-level solutions that would not require changes or special features in the support environment, such as those required in [9,26], which impose changes in the Java virtual machine, hindering portability.

4.4. Dynamic or Adaptive Applications

Some applications have to evolve during operation depending on the varying demands of users in the short or long terms [15]. This capability of evolution could be built by the programmer in an ad-hoc fashion, imposing high development costs and also impairing future component reuse. Most M-LP based approaches do not help or offer clear mechanisms for dynamic adaptation. Configuration programming support environments can offer basic module/connector dynamic reconfiguration capability in flexible ways, using simple support mechanisms. Configuration descriptions are executable and the configuration primitives can be mixed with normal programming code. Alternatively, it is possible to use extended ADLs, that include directives that help the automatic verification of configuration activity consistency [3]. The reconfiguration code can be encapsulated in a configurator module (a meta-level configurator) that receives relevant configuration status events, which may trigger reconfiguration activity. The configuration adaptation concern encapsulation allows the implementation of customized decision-making reconfiguration policies, to cater for different application systems requirements. The actual application architecture reconfiguration can be achieved by changing modules and connectors of the running configuration.

Other schemes for adaptive programming, such as that used in the Darts system [27], 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 forbids 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 requirements of some applications; this was also pointed out in [23].

4.5. Implementation Flexibility

Our ADL includes an Interface Definition Language (IDL), used to specify the modules and connectors interfaces, which uses a syntax compatible with CORBA IDL. The adoption of a standard IDL helps 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 CORBA and Java support environments [17,19]; the Java implementation provides interoperability and component portability in Java environments. 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 [14]); 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 [32].

It is interesting to note that the module implementation strategies identified as useful in [12] can be described by simple language constructs in ADL based proposals. In particular, R-RIO provides a directive to specify the actual module code implementation to be associated with a given interface.

4.6. Comments

ADLs provide a powerful tool to deal with software architectures. However, either in ADL/CP or M-LP environments, it may not be possible the immediate reuse of independently designed modules and connectors (or meta-objects). In some cases, in order to design a connector for a given module set, the designer has to know how the concerned modules interact and even some of the internal details of these modules. This fact is clear in such cases as the observer pattern presented in [13].

Connectors used for communication in distributed systems are “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, exception handling requires the intervention of the application programmer through specific coding. This has to be planned when designing the application, and requires appropriate module internal and external architectures. To summarize, what can be solved at

the meta-level are only limited adaptations. Perhaps, as observed by Saltzer, the treatment of these cases is inherently an application concern [28]. Much more research is necessary to augment reuse in these cases.

5. Conclusions

Configuration programming approaches, based on ADLs, promise many direct benefits for system development and maintenance. The architectural definition naturally pervades all the life-cycle stages of a system, including application-evolving activities. This makes it attractive and feasible to integrate the paraphernalia of languages, tools and mechanisms needed for current software engineering practice in an ADL centered environment, in order to obtain a sophisticated programmer workbench. In particular, the architectural 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; this should help to check if functionally compatible modules will obey explicit interaction contracts. In addition, the use of explicit inter-module connections exposes the information required to guide software reuse.

Configuration programming does not impede the use of particular meta-programming languages to program the configurable components: modules and connectors. At an abstract level, architectural descriptions are orthogonal to implementations and this permits the implementation descriptions to be attached to the architecture at a different stage. In principle, this facilitates the mapping of architectural components to different language concepts and run-time environments. As described in the paper, there is an immediate correspondence between key ADL/CP and M-LP concepts and some of the main issues to be solved for their deployment in real systems are quite similar. According to this view, the adoption of ADL/CP concepts can facilitate the use of diverse existing M-LP approaches by providing simple mechanisms to specify meta-components and meta-architectures through configuration. As a first step towards this goal, we are developing R-RIO, that integrates both approaches in a single, conceptual and practical, configuration programming framework. This experiment should help us to clarify many issues that this work does not directly address.

References

1. Aksit M., Wakita K., et al., Abstracting object interactions using composition filters, in proc. ECOOP'93 Workshop on Object-Based Distributed Programming, pp. 152-184, 1993.
2. Allen, R., Garlan, D., A Formal Basis for Architectural Connection, in *ACM Transactions on Software Engineering and Methodology*, July 1997, also available in www.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/wright/wright_bib.html.
3. 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.
4. Bergmans L.M.J., Composing Concurrent Objects - Aplying Composition Filters for the Development And Reuse of Concurrent Object-Oriented Programming, Ph.D. Thesis, Universiteit Twente, Holland, 1996.
5. Chiba S., A Meta-Object Protocol for C++, in Proceedings of OOPSLA'95, ACM SIGPLAN Notices, Vol. 30, N. 10, pp. 285-299, 1995.

6. Fabre Fabre, J., etal., Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming, in proc. 25th IEEE International Symposium on Fault-Tolerant Computing, 1995.
7. 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.
8. Garlan D., Higher-Order Connectors, presented in Workshop on Compositional Software Architectures, www.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html, 1998.
9. Golm, M.; Design and Implementation of a Meta Architecture for Java, M.Sc. Thesis, Erlanngen-Nüremberg University, Germany, 1997.
10. Helm, R., Holland, I.M. e Gangopadhyay, D., Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, OOPSLA'90, pp. 303-311, 1990.
11. Kiczales G., et al., Aspect-Oriented Programming, in ECOOP'97 Proceedings, Lecture Notes in Computer Science, Springer-Verlag, pages 220-242, 1997.
12. Kiczales G., Lamping J., et al., Open Implementation Design Guidelines, in proc. 19th International Conference on Software Engineering. ACM Press, 1997.
13. Kiczales G., Lopes C., Aspect-Oriented Programming with AspectJ, Xerox PARC, www.parc.xerox.com/aop, 1998.
14. Krieger D., Adler R.M., The Emergence of Distributed Component Platforms, in IEEE Computer Vol. 31, N. 3, pp. 43-53, 1998.
15. Ladaga, R., Veitch, J., Dynamic Object Technology, Communications of the ACM, Vol. 40, N. 5, pp. 37-38, 1997.
16. Lamping J., The Interaction of Components and Aspects, ECOOP'97 AOP Workshop document, www.cs.utwente.nl/aop-ecoop97, 1997.
17. Lobosco, M. R-RIO: A Java Environment for Supporting Evolving Distributed Systems Dissertation (in portuguese), M.Sc. Dissertation, CAA/UFF, Brazil, 1999.
18. Lopes, C.I.V.; D: A Language Framework for Distributed Programming, Ph.D. Thesis, College of Computer Science, Northeastern University, Boston, USA 1997.
19. 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.
20. Loques O., Leite J., Carrera E.V., P-RIO: A Modular Parallel-Programming Environment, IEEE Concurrency, Vol. 6, N. 1, pp. 47-56, 1998.
21. 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.
22. Malucelli, V. V, Babel – Building Applications by Evolution, M.Sc Dissertation (in portuguese), DEE / PUC-RJ, Rio de Janeiro, Brazil, 1996.
23. Matthijs F., et al., Aspects should not die, ECOOP'97 AOP Workshop document, www.cs.utwente.nl/aop-ecoop97, 1997.
24. Mens K, Lopes C.V., Tekinerdogan B, Kiczales G., ECOOP'97 Aspect-Oriented Programming Workshop Report, www.cs.utwente.nl/aop-ecoop97, 1997.
25. Moriconi M., Qian X., Correctness and Composition of Software Architectures, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, U.S.A., pp. 164-174, 1994.

26. Oliva A., Buzato L., Composition of Meta-Objects in Guaraná, in Proceedings of the Workshop on Reflective Programming in C++ and Java, pp 86-90, OOPSLA 98, Vancouver, Canada, 1998.
27. 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.
28. 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.
29. 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.
30. Sztajnberg, A., Flexibility and Separation of Concerns in Distributed Systems, Ph.D. Thesis proposal Dissertation (in portuguese), COPPE/UFRJ, Rio de Janeiro, Brazil, 1999.
31. Sztajnberg, A., Lobosco, M., Loques, O., Configurando protocolos de interação na abordagem R-RIO, XIII Simpósio Brasileiro de Engenharia de Software (aceito para publicação), Florianópolis, outubro de 1999.
32. Wegner P., Interoperability, ACM Computing Surveys, Vol. 28, No. 1, pp.285-287, 1996.
33. 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.