Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio. Architectural Reflection: Realising Software Architectures via Reflective Activities. In Proceedings of the 2nd Int.l Workshop on Engineering Distributed Objects (EDO 2000)*, LNCS 1999, pages 102–115. Springer, Davis, USA, November 2000.*

Architectural Reflection

Realising Software Architectures via Reflective Activities

Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio

D.I.S.Co. – Università di Milano-Bicocca. Milan, Italy
{tisato,savigni,cazzola,sosio}@disco.unimib.it

Abstract. Architectural reflection is the computation performed by a software system about its own software architecture. Building on previous research and on practical experience in industrial projects, in this paper we expand the approach and show a practical (albeit very simple) example of application of architectural reflection. The example shows how one can express, thanks to reflection, both functional and non-functional requirements in terms of object-oriented concepts, and how a clean separation of concerns between application domain level and architectural level activities can be enforced.

1 Introduction

Software architecture is an infant prodigy. On the one hand, it is an extremely promising field, and these days no sensible researcher could deny its prospective importance. On the other hand, it is an extremely immature subject; as a matter of the fact, there is little (if any) agreement even on its definition (see [26] for an extensive, but probably incomplete, list of such attempts).

A major problem in this field is the semantic gap between architectural concepts and concrete implementation. Many architectural issues, in particular those related to non-functional requirements, are realised by mechanisms spread throughout the application code or, even worse, hidden in the depth of middleware, operating systems, and languages' run-time support. This is what we call the implicit architecture problem, which is especially hard for distributed objects systems, where clean objects representing application domain issues rely on obscure system-dependent features related to architectural concepts (static and dynamic configuration, communication strategies, Quality of Service, etc.).

The goal of this paper is to show how a systematic approach based on computational reflection i.e., *architectural reflection*, may help filling this gap by reifying architectural features as meta-objects which can be observed and manipulated at execution time. This lifts up to the application level the visibility of the reflective computations the system performs on its own architecture and ensures a proper separation of concerns between domain-level and reflection-level activities. The proposal derives from the authors' experience both in related research areas and in the development of real, industrial projects where the key ideas presented in the paper have been developed and exploited.

The paper outline is the following. Section 2 presents an example that will be employed as a reference throughout the paper. Section 3 introduces the problem of implicit architecture. Section 4 presents the fundamental concept of architectural reflection. Section 5 sketches, in an ideal scenario, how the requirements of the example can be fulfilled via architectural reflection, while Sect. 6 discusses how the ideas can turn into the practice of real-life systems. Section 7 compares the approach with other work. Finally, Sect. 8 presents the current state of the work.

2 An Example

The following example, which is a simplified version of a real-life problem in the area of on-line trading, will be the basis for the discussion.

In a virtual marketplace one or more feeders (the information providers) provide on-line stock exchange information to a set of customers (the information consumers i.e., basically the clients). Such a system has three basic requirements:

1. the marketplace must ensure that local views of information, held by the clients, are kept aligned with a reference image of the information itself, maintained by the feeders;
2. a stock broker may place buy or sell orders;
3. the marketplace evolves through several different phases. There is an opening phase, during which privileged users (not discussed here) define the initial prices. Then there is a normal phase, which is the only one during which customers may buy or sell. Finally, there is a suspension phase, during which customers can only observe the prices.

The system has three more requirements:

4. customers may dynamically join or leave the marketplace, and an overall supervision of which customers are connected must be ensured. In addition, the marketplace must be able to disconnect a client e.g., for security purposes, should any doubt arise as to the client's actual identity;
5. each customer must be capable of selecting the alignment strategy for their local image: on request (a.k.a. *pull*), on significant changes (a.k.a. *push*), or at fixed time intervals (a.k.a. *timed*);
6. the system must be flexible with respect to the number and physical deployment of the feeders. Adding or removing a feeder should *not* imply any change in the client code.

All this looks quite simple. However, as soon as we go through the analysis and design process (for instance using UML [3]), we recognise that requirements 1, 2, and 3 (let us call them "functional requirements") can be easily expressed in terms of well-defined domain classes. 1 and 2 can be expressed via simple class and interaction diagrams, and 3 via state diagrams.

Things are not as simple for requirements 4, 5, and 6 (let us call them "non functional requirements"). Regarding 4, any distributed platform provides mechanisms for dynamic connections to services. However, in most cases it is not easy to monitor who is connected. Such information exists somewhere inside the middleware, but it can hardly be observed and relies on platform-dependent features. The result is that the application is not portable. We would like to provide the end user (e.g., the manager of the marketplace) with a clean and platform-independent visibility of the status of the connections.

A similar, but harder, problem holds for requirement 5, whose fulfilment implies the definition of an application-level protocol whose behaviour can be dynamically selected according to users' taste. Even if at the analysis stage the protocol is well specified (e.g., as a state machine), at the design and implementation stages it is split into a specification of individual components' behaviour and then implemented by components' code. This code (implementing an architectural choice) will be intermixed with architecture-independent, functional code. In current practice, most architectural choices follow this fate and get dispersed in the components' code in implemented systems. Moreover, the implementation relies on elementary transport mechanisms (RPC or asynchronous messages or the like). Therefore, a clean and platform-independent visibility of the communication strategies is not provided at the application level.

Finally, point 6 is just an example that raises the most important issue dealt with in this paper i.e, the implicit architecture problem, whereby information about the system architecture is embedded in application-level code. As a matter of the fact, the number and actual deployment of the feeders is an architectural issue, that should *not* be embedded in application-level code.

3 Implicit Architecture

The example highlights that, while object-oriented technology provides a sound basis for dealing with domain-related issues, we still lack adequate notations, methodologies, and tools for managing the *software architecture* of the system [24]. The architecture of a software system based on distributed objects is defined by stating:

- how the overall functionality is partitioned into *components*;
- the *topology* of the system i.e., which *connectors* exist between components;
- the *strategy* exploited in order to co-ordinate the interactions and, in particular, how connectors behave.

Existing notations for composing software modules are usually limited to expressing some small subset of these issues e.g., topology alone or basic communication paradigms such as RPC or asynchronous messages. Many architectural concepts, even if well specified at the design stage, are no more clearly visible at the implementation stage and at execution time. Moreover, many architectural issues (such as scheduling strategies, recovery actions, and so on) are tied to the concrete structure and behaviour of the underlying platform. Therefore, such issues

are spread throughout the (opaque and platform-dependent) implementation of OS and middleware layers. In other words, this means that these concepts are confined at the *programming-in-the-small* level [8].

Ultimately, there is no separation of concerns among domain, implementation, and architectural issues, and the latter ones are not clearly visible and controllable at the application level. This is what we termed the *implicit architecture problem (IAP)* in a previous paper [5].

Designing and building systems with implicit architectures has several drawbacks: most notably, it hinders components' reuse due to the architectural assumptions components come to embed [10]; it makes it infeasible to reuse architectural organisations independent of the components themselves; it makes it overly complex to modify software systems' architecture; and it is also cause of the undesirable, yet empirically observed fact that architectural choices produced by skilled software architects are most often distorted and twisted by implementers [15]. A more detailed discussion of the IAP and its consequences can be found in [4].

The IAP is especially serious because in most real systems architectural issues must be dealt with at execution time. The topology may dynamically change either by adding new components or by modifying their connections, in order to extend or modify system functionality, to add new users, to enhance availability, to perform load sharing, etc. The strategy too may change in order to meet changing user requirements and timing constraints, to ensure a given average rate and reliability for data transfer, and so on. The need for dynamic management of architectural issues often arises from non-functional requirements i.e., configurability, availability, performance, security and, in general, Quality of Service.

Figure 1 sketches a typical situation in terms of concrete run-time architecture (i.e., in terms of well distinguished objects which exist at execution time). Even if components exist at run-time as (distributed) objects, the connectors' implementation is heavily spread. Inside the components there are code fragments which rely on application-level interfaces such as programming language constructs, middleware APIs, or OS primitives. Such interfaces are implemented inside the underlying platforms via hidden mechanisms which, in turn, interact in mysterious ways. Unfortunately, such mechanisms implement architectural issues (mainly those related to non-functional requirements) which can be hardly observed or controlled.

Expressing functional requirements in terms of application domain concepts modelled as classes and objects provides the basis for building well-structured systems which exploit domain-level classes and objects to fulfil the functional requirements. Accordingly, *expressing non-functional requirements in terms of architectural concepts modelled as classes and objects should be the basis for building well-structured systems which exploit architectural classes and objects aimed, in particular, at fulfilling non-functional requirements.* As a matter of the fact, the IAP mainly arises from the fact that architectural information is

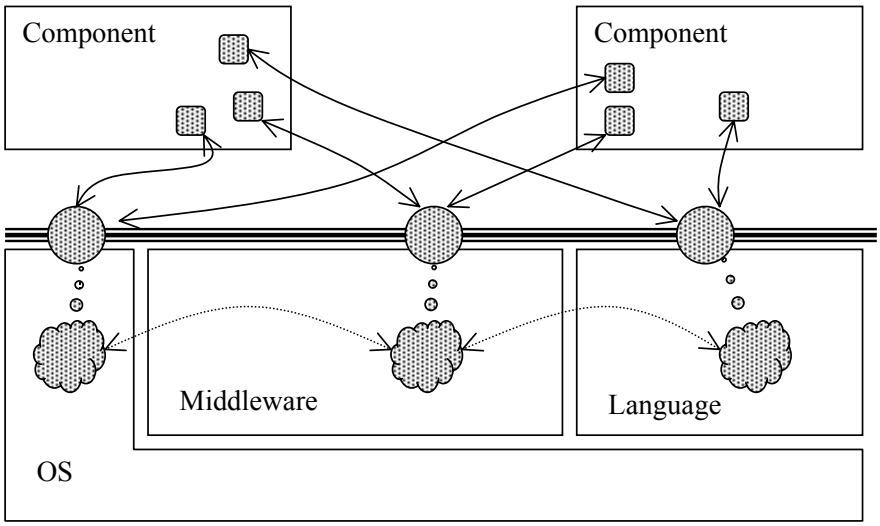

Fig. 1. The Implicit Architecture Problem

spread throughout application and platform code, and is not properly modelled in terms of well-distinguished classes and objects.

4 Architectural Reflection

All of the above issues imply that a running system performs computations about itself. More practically, there exist somewhere data structures representing system topology and system behaviour, plus portions of code manipulating this information. Architectural reflection (AR) basically means that there exists a clean self-representation of the system architecture, which can be explicitly observed and manipulated.

In its general terms, computational reflection is defined as the activity performed by a software agent when doing computation on itself [14]. In [4, 5] we defined architectural reflection as the computation performed by a system about its own software architecture. An architectural reflective system is structured into two levels called architectural levels: an *architectural base-level* and an *architectural meta-level*.

The base-level is the "ordinary" system, which is assumed to be designed in such a way that it does not suffer from the IAP; in the sequel we shall discuss how to achieve this goal. The architectural meta-level maintains causally connected objects reifying the architecture of the base-level.

According to the concept of domain as introduced in [14], the domain of the base-level is the system's application domain, while the domain of the architectural meta-level is the software architecture of the base-level.

Architectural reflection can be further refined into *topological* and *strategic* reflection. Topological reflection is the computation performed by a system about its own topology. With regard to the example, topologically reflective actions include adding or removing customers to the marketplace.

Strategic reflection is the computation performed by the system about its own strategy; for example, dynamically changing the strategies for the alignment of customers' local images of the stocks.

Architectural reflection has the desirable effect of lifting architectural issues up to the programming-in-the-large level.

5 An Ideal Scenario

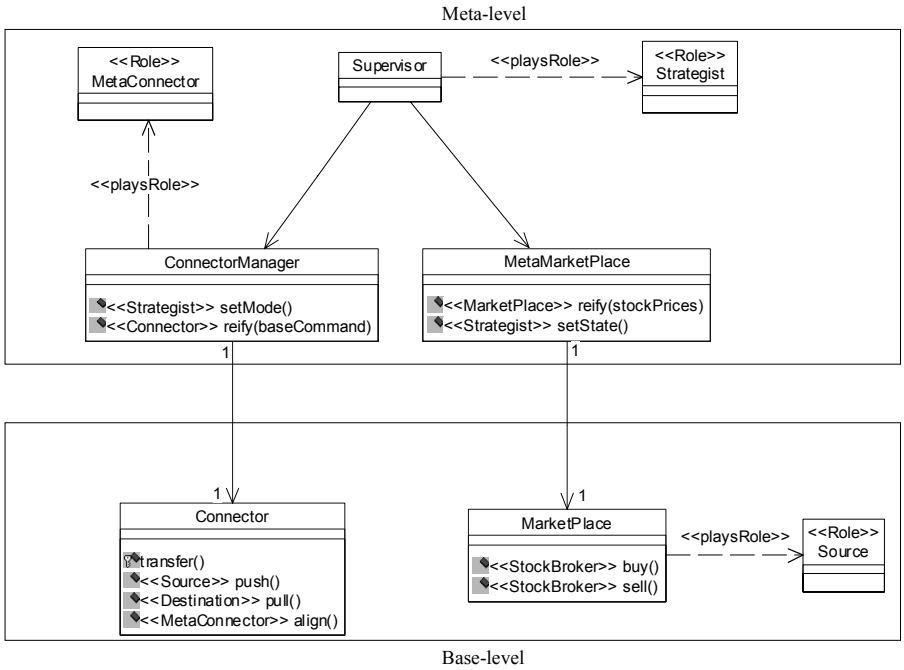

Fig. 2. An example of an ideal system

Architectural reflection is quite a general concept, and assumes only a IAP-free base-layer. This section briefly describes the ideal approach in building both the base- and the meta-level. The on-line trading example first encountered in Sect. 2 will be used as a (simplistic) case study (see Fig. 2).

Note that in this example, roles are treated as first-class entities, and represented as classes with a `Role` stereotype. A class can play one or more roles

(via the `playsRole` stereotype). Operations can be restricted to one or more roles only. This means that only instances of classes playing that role can call those operations. This selective operation export is represented by tagging the operation with the stereotype bearing the name of the role the operation is exported to (e.g., `«Source»`). In some cases, such as operation `reify()` in class `ConnectorManager`, the operation is directly exported to class `Connector` for the sake of simplicity; however as a general rule, operations are never exported directly to object classes in order to avoid hard-coding in a class the dependency to a particular context (see [19] for further details).

5.1 Base-level

Both components and connectors in the base-level should be realised as passive entities. This means that they should not embed any activation strategy. In this way, they can be reused under completely different strategies.

Connectors embed all communication issues, such as relationships with the underlying middleware (if any), network protocols, and so on. They export to the above levels a uniform interface, which is strictly independent on the underlying implementation details. In this way, when composing a system one can reason at a higher abstraction level than that allowed for by common middleware (such as CORBA), in that one can actually ignore distribution issues. This is not the case when interacting directly with middleware.

In other words, the goal of our approach is to *separate* distribution from the other issues, definitely not to ignore distribution. This is a very controversial point, about which a lot of discussion is taking place (see for example [11]). One very popular approach (adopted for example by CORBA) is to *mask* distribution; every method call is accomplished in the same way regardless of the actual location of the target object (this is the largely touted "distribution transparency"). This approach has the undisputed advantage of greatly fostering reuse, as components are independent of distribution. However, many people argue that issues such as latency and partial failure, typical of distributed systems, make it impossible in practice to systematically ignore distribution, especially in real-time systems.

We believe that many of these problems stem from an excessive urge to encapsulate. While encapsulation is undoubtedly a key achievement in software engineering, it should not be abused. Issues that are of interest to the rest of the system should definitely be visible through the encapsulating shield. In addition, in some cases distribution is meaningful even at the analysis level (the so-called "intrinsic" – or should we call it "analysis-level" – distribution, as opposed to "artificial" – or should we call it "architecture-level" – distribution useful e.g., for fault tolerance purposes). All of this can be ascribed to the very in issue of Quality of Service, that we do not mean to address here. Our approach to distribution tries to separate distribution from other issues, not to ignore it. If distribution is kept well separate from other issues (computation, strategy, etc.), it can be properly dealt with in one place and ignored by the rest of the system.

In the example, the `MarketPlace` class exports only two operations, both to the `StockBroker` role (in the base-level). The idea is that only application-domain activities are represented at the base-level. So for example, in the `Marketplace` class there is only information about the current value of stocks, and there is no way at this level to influence the policies of the stock market (such as opening or closing the negotiations).

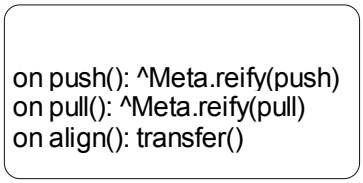

Fig. 3. The state diagram for class `Connector`

Similarly, the `Connector` class only offers the means to push and pull information; the corresponding operations are exported to the `Source` and `Destination` roles respectively. Consequently, the corresponding state diagram is trivial[1](see Fig. 3).

The `align()` operation can only be called from the meta-level, and is implemented by the protected `transfer()` method, which actually performs the information transfer.

5.2 Meta-level

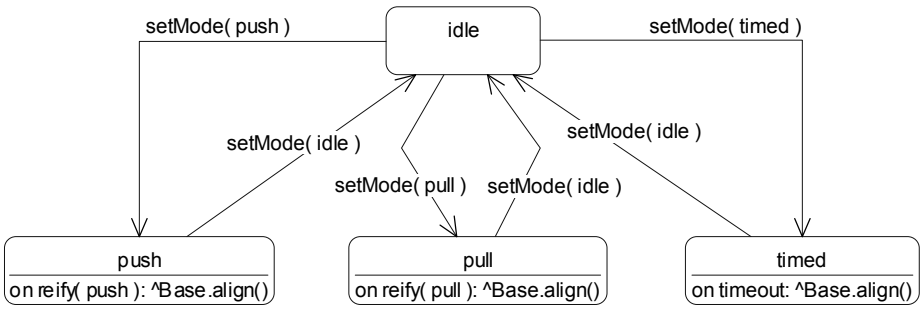

Fig. 4. The state diagram for the `MetaConnector` class

[1] Due to space limitations, throughout the example we only detail communication-related classes.

In the meta-level reside meta-objects for both components and connectors. These encapsulate the *policies*, both application-related and communication-related. So for example, the `MetaMarketPlace` class encapsulates all the policies that are usually enforced by the financial authorities (such as withdrawing stocks from the market). Similarly, the `MetaConnector` class manages the communication policies (push, pull, timed), as shown in Fig. 4.

In addition, in order to meet requirement 4 in Sect. 2, a topologist must be introduced that manages, at execution time, the architecture of the software system, by instantiating components and/or connectors. For further details, see [6, 7].

6 Turning Ideas into Practice

The approach outlined in the previous chapter, albeit simplistic, sets the ideal properties of an architectural reflective system. However, it is widely known that a software system is almost never built from scratch; more often than not, the prevalent activity in software construction lies in integrating existing pieces of software into a new product. Therefore, no discussion of a practical software construction paradigm would make any sense without turning to the real world and discussing the practical applicability of the paradigm.

Architectural reflection is no exception; in this section we briefly set forth some requirements that existing systems must meet in order to be good candidates for being integrated into an architectural reflective system.

6.1 COTS

COTS should not embed architectural issues. Apart from architectural reflection, this seems to be a general requirement (see also [10]). In other words, COTS should simply provide functionality without falling into the IAP.

For them to be part of an architectural reflective system, they should provide:

- hooks for connectors. Such hooks should be visible in the component's interface;
- visibility of their internal state, so as to allow meta-level entities to operate on them;
- most important, they should embed no activation strategy. In other words, they should be passive entities.

6.2 Legacy Systems

It is often said that any running system can be thought of as a legacy system. Due to this variety, integrating legacy systems into a new product is often close to impossible. Such systems are more often than not poorly documented or not documented at all, and are often built out of spaghetti code. In these systems it is often extremely hard even to understand *what* the system does, let alone *how* it does it.

The solution usually employed to integrate this class of systems is to wrap them in a modern, most often object-oriented shield (this is for example the approach used to integrate them into a CORBA environment). However, since it's hard to fully grasp their functionality, usually just a subset of this is actually exposed by the shield.

Having said that, the most basic requirement for integrating legacy systems is that systems should embed no activation strategy. Since this can be an overly strong requirement, it can be weakened into the following: embedded strategies are allowed as long as they do not affect domain requirements. This basically means that embedded strategies should not prevent an external strategist to *examine* the basic policies and *modify* them when needed. If the system is completely opaque and allows no insight into its strategy, then it just cannot fit into an architectural reflective system.

6.3 One Remark About Conceptual and Practical Approaches

Sometimes it may be impossible or impractical to build an actually layered system such as the one outlined in Sect. 5. However, it should be noted that, even when *actually building* such a system is impossible, one can always follow the *conceptual approach* described in this paper, possibly implementing the system using conventional means.

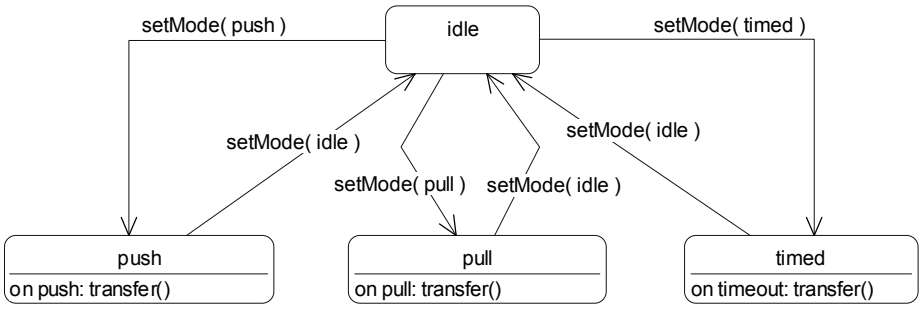

Fig. 5. The state diagram of a merged `Connector` class

So for example it may be impractical to implement connectors with two classes (`Connector` and `ConnectorManager`, see Fig. 2). In that case, one can merge the two classes into one, obtaining a state diagram such as the one shown in Fig. 5.

In this case, there is no explicit distinction between the two layers. However, a nice separation of concerns between domain-related activities (information transfer) and architectural activities (selection of policies) still holds; in fact the `on xxx: yyy()` clauses represent the former, while actual state transitions represent the latter.

In other words, a conceptual approach is useful even if it does not turn into actual design and code. This is like implementing a well-designed, object-oriented system in assembly language for optimisation purposes; the level of reuse is certainly not the same as using an OO language, but most of the benefits of a good design are preserved.

6.4 One Remark About Architectural Languages

A lot of discussion is going on in the software architecture community about what is to be considered an architectural language and what isn't. As a matter of the fact, OOPSLA 99 hosted a Panel session entitled "Is UML also an Architectural Description Language?"

Clearly, in the conventional, "flat" (i.e., non-reflective) approach neither design languages nor programming languages offer sufficient means to express architectural issues. Thus, the need arises to extend the programming paradigms with architectural/non-functional issues. Examples include introspection à-la JavaBeans, communication primitives with timeout, priorities, etc.

As far as the UML in particular is concerned, the answer to the above question is certainly: "no, using the flat approach. Yes of course, provided a reflective approach is employed instead." As a matter of the fact, one of the key features of reflection is that the same language can be used for both base-level and meta-level, provided that the language includes a mechanism (reification and reflection) for causal connection. In (architectural) reflective terms, switching from domain level to architectural level simply means changing domain (again in the sense used in [14]). This does not imply a change in language. In other words, architectural objects can be treated as first class objects (architectural objects) in the reflective level, thus achieving the fundamental goal of separation of concerns.

7 Related Work

The idea of enforcing a separation of concerns between basic computational blocks and the entities governing their overall behaviour and cooperation periodically reappears in different branches of information technology under different guises and formulations. The seminal paper by DeRemer and Kron [8] proposed using a different notation for building modules and for gluing modules together, yet this latter notation could only convey simple define/use relationships. A whole family of coordination languages, termed control-driven [17], and especially the Manifold language [2], are also based on the idea of separating computation modules (workers) from "architectural" modules (managers) at run-time. While this is quite close to the concept of explicit architecture on which this proposal builds, it is not the intent of Manifold (and other coordination languages) to allow the encapsulation of architecture in the broadest sense of the term. Nevertheless, it must pointed out that Manifold also provide constructs to address dynamic architectural change via managers, which also has similarities

with AR. Other recent proposals focus on run-time connectors (explicit run-time representation of cooperation patterns) and include Pintado's gluons [18], Aksit et al.'s composition-filters [1], Sullivan's mediators [25], and Loques et al.'s R-Rio architecture [13].

Several authors have confronted with the problem of modifying architecture at run-time, for reconfiguration or evolution purposes. Kramer and Magee [12] discuss an approach to runtime evolution that separates evolution at the architectural and application level. Architectural reconfiguration is charged to a configuration manager that resembles AR's meta-entities. In their approach, nevertheless, the meta-level has a limited visibility of the "base-level" state (i.e., it only perceives whether base-level entities are in a "quiescent" state). Oreizy et al. [16] propose a small set of architectural modification primitives to extend a traditional (non-dynamic) ADL, and also exploit connectors to let architectural information be explicit in running systems. With respect to AR, their approach is more related to defining what operations are useful at the meta-level than to devising how the meta-level and base-level should interact (which is the main focus of this paper).

Very few works insofar have pointed at the advantages of a reflective approach for the design of systems with dynamic architecture. One of those few proposals is that of Ducasse and Richner, who propose introducing connectors as run-time entities in the context of an extended, reflective object model termed FLO [9]. FLO's connector model is very rich and interesting, and has several similarities to ours. Nevertheless, FLO is based on a simpler component model which does not include a behavioural component specification.

8 State of the Work

The approach described above is far from a speculative vision. On the contrary, it's being employed under many forms in several practical situations.

A somewhat simplified version of the framework is the base for the Kaleidoscope reference architecture [21–23], which is being employed in several industrial projects in the areas of traffic control and environmental monitoring. Kaleidoscope is also undergoing major improvements in the form of an object-oriented framework and an associated methodology, all heavily influenced by the reflective concepts described above.

The idea of separating functional from non-functional, and in particular management, issues has been successfully exploited in the design and implementation of a platform which integrates heterogeneous devices and applications for traffic control at the intersection level [20].

On a more theoretical, long-term perspective, we are working on a more formal definition of reflective architecture. In particular we are evaluating UML (possibly augmented with OCL [27]) as a meta-level architectural language, which is giving sound and promising results. In particular, the concept of role as described in Sect. 5 is proving very interesting and powerful (see [19] for further details).

References

[1] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of Object-Based Distributed Programming (ECOOP94 Workshop)*, number 791 in LNCS, pages 152–184. Springer-Verlag, Jul 1994.

[2] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and Its Implementation. *Concurrency: Practice and Experience*, 50(1):23–70, Feb 1993.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. A Fresh Look at Programming-in-the-Large. In *Proceedings of The Twenty-Second Annual International Computer Software and Application Conference (COMPSAC '98)*, Vienna, Austria, Aug 13-15 1998.

[5] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of the 2^{nd} Euromicro Conference on Software Maintenance and Reengineering and 6^{th} Reengineering Forum*, Florence, Italy, March 8-11 1998.

[6] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural reflection: Concepts, design, and evaluation. Technical Report RI-DI-234-99, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1999.

[7] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of Automated Software Engineering – ASE'99 14^{th} IEEE International Conference*, pages 263–266, Cocoa Beach, Florida, USA, Oct 12-15 1999.

[8] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus Programming-in-the-small. *Transactions on Software Engineering*, SE-2:80–86, June 1976.

[9] Stéphane Ducasse and Tamar Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of ESEC'97*, LNCS 1301, pages 483–500. Springer-Verlag, 1997.

[10] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of XVII ICSE*. IEEE, April 1995.

[11] Rachid Guerraoui and Mohamed E. Fayad. OO Distributed Programming Is *Not* Distributed OO Programming. *Communications of the ACM*, 4(42):101–104, 1999.

[12] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transaction on Software Engineering*, SE 16(11), Nov 1990.

[13] Orlando Loques, Alexandre Sztajnberg, Julius Leite, and Marcelo Lobosco. On the Integration of Configuration and Meta-Level Programming Approaches. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 191–210. Springer-Verlag, Heidelberg, Germany, June 2000.

[14] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA87, Sigplan Notices*. ACM, October 1987.

[15] Gail C. Murphy. Architecture for Evolution. In *Proceedings of 2nd International Software Architecture Workshop*. ACM, 1996.

[16] Per Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings ICSE 98*, pages 177–186, Kyoto, Japan, April 1998.

[17] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. *Advances in Computers*, 46, 1998.

[18] Xavier Pintado. Gluons: A Support for Software Component Cooperation. In *Proceedings of ISOTAS'93*, LNCS 742, pages 43–60. Springer-Verlag, 1993.

[19] Andrea Savigni. RoleJ. A Role-Based Java Extension. In *Proceedings of ECOOP 2000 (The 14^{th} European Conference for Object-Oriented Programming)*, Cannes, France, June 12 – 16 2000. Poster. To appear.

[20] Andrea Savigni, Filippo Cunsolo, Daniela Micucci, and Francesco Tisato. ESCORT: Towards Integration in Intersection Control. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8^{th} Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 11 – 14 2000.

[21] Andrea Savigni and Francesco Tisato. Kaleidoscope. A Reference Architecture for Monitoring and Control Systems. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, USA, February 22-24 1999.

[22] Andrea Savigni and Francesco Tisato. Designing Traffic Control Systems. A Software Engineering Perspective. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8^{th} Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 11–14 2000.

[23] Andrea Savigni and Francesco Tisato. Real-Time Programming-in-the-Large. In *Proceedings of ISORC 2000 The 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 352 – 359, Newport Beach, California, USA, March 15 – 17 2000.

[24] Mary Shaw and David Garlan. *Software Achitecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[25] Kevin J. Sullivan, Ira J. Kalet, and David Notkin. Evaluating the Mediator Method: Prism as a Case Study. *IEEE Transactor on Software Engineering*, 22(8):563–579, August 1996.

[26] Software Engineering Institute Carnegie Mellon University. How Do You Define Software Architecture? `http://www.sei.cmu.edu/architecture/definitions.html`.

[27] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1999.