# Constraining Architectural Reflection for Safely Managing Adaptation

Rui S. Moreira
*University Fernando Pessoa* *Lancaster University* *INESC Porto*
*moreirar@ufp.pt* *moreirar@comp.lancs.ac.uk* *rjm@inescporto.pt*
Gordon S. Blair
*Computing Department - Lancaster University*
*gordon@comp.lancs.ac.uk*
Eurico Carrapatoso
*Faculty of Engineering - University of Porto* *INESC Porto*
*emc@fe.up.pt* *emc@inescporto.pt*

## Abstract

Architecture evolution is typically a requirement of modern and pervasive software systems. In particular, distributed applications, such as digital libraries and learning systems, due to their interactive and ubiquitous nature, experience contextual forces that induce the need for secure adaptation solutions. Hence, the need for flexible component models that are able to cope with the plethora of multimedia applications, interaction devices and networks widely utilized by mobile end-users. In this paper we present *FORMAware*, a reflective component framework that explicitly provides architecture awareness for constraining and safely managing middleware reconfiguration.

## 1. Introduction

Prevalent middleware solutions provide only design-time reconfiguration facilities usually based on minimal reflective mechanisms (e.g. smart proxies, interceptors, containers, custom marshalling) [12, 16] that have limited and particular applicability. These interception-based mechanisms introduce a degree of flexibility for choosing different management strategies and for adapting the middleware behaviour to specific contextual applications. Nevertheless, these approaches are not generic and open enough to cope with broader adaptation scenarios since they only allow static and pre-defined reconfigurations (declared at design time) and do not support, for example, dynamic policy switching. Furthermore, in our opinion, the most important shortcoming of current component models (in terms of dynamic adaptation) is the lack of abstractions to explicitly represent distributed composites, connectors and architectural constraints. In addition, the available meta-information facilities are currently limited to component types and local component dependencies (e.g. local bindings) that only address per-component evolution. Therefore, we feel it is important to derive new middleware architectures in order to cope with both short and long-term (i.e. predictable or unpredictable) adaptation requirements [9]. We plan to do this by combining architectural awareness and reflection.

Reflection focuses on increasing flexibility and providing arbitrary levels of openness. In fact, this is usually pointed out as the *Achilles heel* of reflective systems, i.e. the lack of safety bounds for preventing unconstrained adaptation, hence, system malfunctions. Some studies (although not many) address this important issue of safe/constrained adaptation. More specifically, dynamicTAO [6] and K-Components [2] tackle architecture awareness via explicit representation of component dependencies. Nevertheless, none of the solutions advocates the use of an explicit style manager with explicit representation of style rules. In addition, the representation and description of component dependencies is usually limited to local/direct bindings. Equally important is the fact that some of the major issues behind adaptation (e.g. state transference between replaced components, synchronisation of reconfiguration with the running system) [13] are usually neglected or secondarily considered. Our position is that a solution may be in extending architectural reflection to capture domain specific semantics and use it for safely govern architecture adaptations.

We recognize that the analysis and evaluation of systems architectures is undoubtedly facilitated by the use of *Architecture Description Languages* (ADLs) which promote, for example, explicit assembly and late wiring, topology awareness and style constraining [11]. However, current ADL frameworks [8, 14] neither explicitly represent architecture style rules (as first class objects) nor do they integrate them with an explicit architecture manager. Moreover, there is no visible and principled relationship between the entity controlling

the reconfiguration (e.g. adaptation manager) and the meta-information constraining the adaptation. We argue that embedding the architecture representation into the programming model promotes the proximity between design and development, thus raising the flexibility of middleware. We argue that it is possible to bring in the best practices from software architecture [11] for managing run-time reconfiguration. However, none of the current ADL-frameworks provides such an open and integrated architecture awareness approach.

In effect, what we are proposing is to extend the seminal idea of combining *object-oriented* techniques with *open-implementation* designs to raise the levels of abstraction and encapsulation (for tackling complexity) and also increase the openness of software (for enhancing controllability) [5]. We take a step beyond this by combining *design patterns* and *reflection*, the former for capturing design knowledge and collaborations at different levels of scale and abstraction while the latter for raising the levels of flexibility and adaptability. Our approach advocates and encourages the coalition of these two concepts for achieving the component-based software development paradigm. More specifically, we promote the combination of design patterns and reflective techniques to provide a customizable and extensible solution (cf. component framework) for supporting the specification and safe management of adaptation. Our solution is prototyped in *FORMAware*, a reflective component-based framework that opens the architecture domain (via introspection and adaptation meta-objects) and uses the architecture semantic for safely and transparently (via a transaction-based service) manage the reconfiguration of distributed systems. This paper presents the architecture details of *FORMAware* (section 2) and then finishes with a brief conclusion (section 3).

## 2. *FORMAware* architecture in detail

### 2.1. Overview

The *FORMAware* programming model distinguishes components from connectors, respectively represented by the *BasicComponent* and *BasicConnector* classes, the former used for performing computation tasks and the latter dealing with communication aspects. Both components and connectors possess certain properties and interfaces. However, a distinction is made between provided and required interfaces that are respectively represented by the *ProvidedInterface* and *RequiredInterface* classes. Provided interfaces are *proxies* [3] to the implementation object of the component while re-

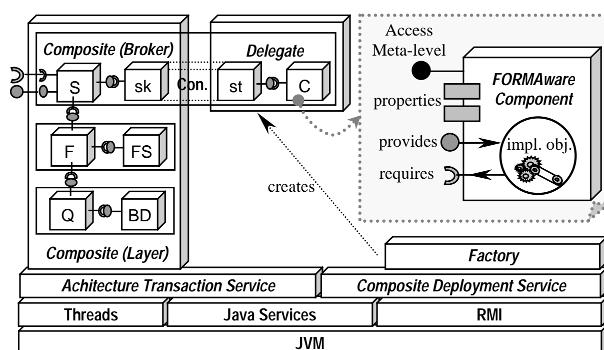

**Figure 1: architecture of the *FORMAware* prototype.**

quired interfaces operate as *placeholders* for provided interfaces offered by other components (see figure 1). *FORMAware* treats these architecture elements as first class objects, thus allowing the explicit and late assembly of components that can be made independently from the development of individual components. Consequently, composite components are also first class elements, each possessing an architecture which is responsible for the configuration of components according to a specific architecture style manager (e.g. *Layer* and *Broker*, as represented in figure 1). The programming model is then extended *with* reflective meta-objects that expose the content and topology of both basic and composite components (section 2.2 details the structure of the meta-level). For an extended description of the programming model, please confer [9, 10].

### 2.2. Using architecture reflection for constraining adaptation

#### 2.2.1. Structuring the meta-level

Different meta-objects shape the structure of the *FORMAware* meta-level. More specifically, both basic and composite components inherit from the *RComponent* class which provides methods for creating specific introspectors and adapters (cf. *method factory* design pattern [3]). These meta-objects provide specific *Meta-Object Protocols* (MOPs) for inspecting or changing the details of each architecture element (i.e. access the *interface* or *architecture meta-models* [1]). We may, for example, discover provided and required interfaces of components, set/get properties, browse the graphs of composites, check their architecture manager and style rules). More interesting is the design defined for opening the *architecture meta-model* that allows us to explicitly assemble components (*awareness of composition*), specify the type of topology (*awareness of structure*) and the rules governing it (*awareness of constraints*), as explained below.

#### 2.2.2. Awareness of composition (explicit wiring)

In *FORMAware*, the assembly of components is explicitly represented by composite components (*RComposite*) which are configurations of architecture elements (i.e. components and connectors) plugged together through their provided and required interfaces. Composites can also export properties and interfaces, provided or required by internal components. Each composite, similarly to basic components, has an implementation object that executes the configuration algorithm responsible for creating the instances of internal components and then plugging and deploying them, to build the composite architecture. The topology of each composite is shaped by a specific architecture strategy as described next.

#### 2.2.3. Awareness of structure (architecture style)

In contrast to other middleware solutions, the composition model proposed by *FORMAware* is not flat, i.e. there is an explicit *architecture* graph (represented by the *RArchitecture* class) which is governed by a *style manager* [9]. The style manager (*polymorphically* represented by the *StyleManagerI* interface) corresponds to the *strategy* (cf. *strategy* design pattern [3]) used for constraining the architecture *context* (i.e. architecture graph) in which the architectural operations are invoked/performed (see figure 2). By default, the style manager is *DefaultArchitectureStyle*. However, developers may define and set a different style (e.g. *Layer*, *PipeFilter*, *Broker*, *Blackboard*) for meeting specific architecture requirements. Afterwards, it is possible to create an adapter meta-object (cf. *AdapterRArchitecture*) for invoking architecture operations on the composite component (e.g. add and plug components, establish dependencies between afar components, such as an *encrypt* and counterpart *decrypt* component, etc.).

Each operation invoked on the adapter is passed to the equivalent *architecture operation* (in the architecture context) which forwards it to an associated *stylish operation* (on the style manager) - see figure 2. Before calling-back the *primitive* graph-operation on the architecture graph, the *stylish* operation executes a series of style *checks*. These verifications follow a certain *template*, according to the type of architecture operation and groups of rules (as described in the next section) used by the style manager (cf. *template method* design pattern [3]). For example, a *stylishPlug* operation, which locally binds 2 components, firstly has to call several *check* methods (e.g. verify the existence of components, provision/requisition of interfaces, location of components, compatibility of interfaces) and, secondly, execute the *primitive plug* method that calls-

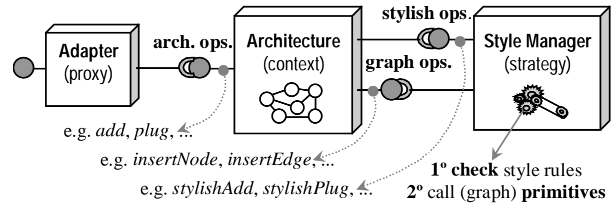

**Figure 2: structure of the architecture meta-model.**

back the *insertEdge* operation on the architecture graph (to effectively plug the two components). This specific combination of design patterns (cf. *strategy* and *template method*), used for structuring the architecture meta-model, enables us to customise architecture operations, i.e. re-order or even re-define the style checks and primitives performed for each architecture operation. Consequently, developers may define new style extensions by sub-classing and re-implementing the default architecture style.

#### 2.2.4. Awareness of constraints (style rules)

An architecture style specifies an ontology of terms and a set of rules that restrict the types of architecture elements and possible configurations [11]. In *FORMAware*, these rules are explicitly represented by *StyleRule* classes, each providing a *verifyValidity* method for checking specific architecture constraints [9, 10]. For example, table 1 illustrates the groups of style rules defined for the default style manger. Developers may extend these groups with new rules for enforcing additional style checks.

The style rules are characterised by their type (i.e. group they belong to) and also the type of architecture operation they apply to. The style manager uses this information to select (based on a mechanism of *guarded execution*) the set of rules that must be checked for each architectural operation. These rules impose conditions that enforce the style and minimise integrity breakdowns, i.e. define an architecture semantic that must be satisfied by all invoked architectural operations, otherwise they may not be committed by the transaction service (described next).

## 2.3. Using the transaction service for managing the adaptation process

#### 2.3.1. Architecture of the service

Any architectural reconfiguration entails the danger of inconsistency. Therefore, to guarantee the consistency of the final configuration, we provide the means to check architecture constraints (as seen above) and also synchronise the overall process of reconfiguration [10]. More specifically, *FORMAware* proposes a

| Groups of rules | Examples of checks validated by the rules |
|---|---|
| Component Model | number of interfaces that each component must provide to and require from other components; number of operations and arguments that, respectively, each interface and operation must provide/possess; |
| Components & Connectors | types of components and connectors in accordance with the style in use (e.g. PipeFilter style uses the Filter type for components and the Pipe type for connectors); |
| Properties | existence of certain properties (associated with the components) and the limits on the values they can store (e.g. maximum and minimum values allowed); |
| Location | location of components (specially important for assuring the existence of components and avoiding direct plugs between remote components); |
| Interfaces | types of interfaces and their signatures (e.g. default architecture style uses the two generic ProvidedInterface and RequiredInterface types); |
| Relationships (Symbiosis) | existing dependencies or symbioses between components that we may need to remove or replace (e.g. between a coder and decoder or the server and client parts of a connector ); |
| Plugs | existence (in the specified components) and compatibility of required and provided interfaces; types of interfaces and types of components being plugged; existence of previous binds; check the state of components; |
| Provision | existence and the type compatibility of the interfaces being exported or check the binds on that specific interface before un-exporting it; |
| Requisition | possibility to import a certain type of interface belonging to an internal component or the existence of external components plugged to the interface that we need to un-import; |

**Table 1: groups of rules checked/validated by the default style manager.**

transactional reconfiguration process that permits us to initiate, combine, commit and rollback architecture operations. The design of the service is inspired by a basic scheduling architecture (cf. *locking scheduler* [4]) and also by a behavioural pattern (cf. *active object* design pattern [7]), both adapted for managing the concurrent nature of the adaptation process. The service possesses a *transaction manager* that creates (on-demand) architecture *transactions* and *enlists* them on the *lock manager*. A *resource manager* is also created (for each initiated transaction) and used by clients for invoking the architecture operations in the scope of the transaction. For each enlisted transaction, the lock manager creates a *scheduler thread* that is responsible for scheduling the architecture requests. Every architecture operation is wrapped by a *request* object that is passed (via *put*) to the associated transaction. The scheduler thread is responsible for *getting* these requests, interpreting them and, then, *generating* specific requests to be *enqueued* in the request stream of the lock manager. For example, figure 3 shows that the scheduler *enqueues* several requests (cf. *lock*, *clone*, *add*, *unplug*, *plug*, *remove*, *unlock*, *switch*) to be dispatched in the scope of a *replace*. These intermediate requests depend on the selected scheduler policy (described in next section). The lock manager is also responsible for initiating a unique *dispatcher thread* that uses a *round-robin* policy for *dequeuing* and *executing* the transaction requests generated by all schedulers.

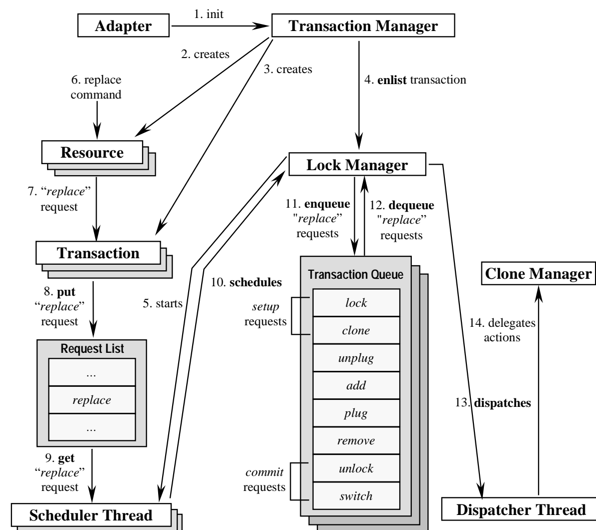


**Figure 3: transaction service - requests for a "*replace*".**

### 2.3.2. Adaptation policies

When initiating a transaction, developers must choose the adaptation policy (e.g. *SKIP, WAIT or FORCE* safe state) that determines the scheduler type (e.g. *SchedulerSkipSafeState*, *SchedulerWaitSafeState* or *SchedulerForceSafeState*). Each scheduler uses specific *setup* and *commit-time* action requests. For example, the *SchedulerSkipSafeState enqueues* a *lock* and *clone* requests before any other actions. Moreover, for committing a transaction it *enqueues* the *unlock* and *switch* requests, respectively for *unblocking* and *commuting* the adapted components. In the same way, the schedulers *SchedulerWaitSafeState* and *SchedulerForceSafeState* implement their own locking and switching strategies. For example, at *setup-time*, the former *enqueues* the *wait*, *lock* and *clone* requests while the latter *enqueues* the *force*, *lock* and *clone* actions (before any other requests) for enforcing a specific adaptation strategy. Similarly, at *commit-time* each scheduler *enqueues* final requests for *transferring the state* of components and *synchronising* the reconfiguration (e.g. *transfer, switch*) before committing the adaptation process.

## 3. Conclusion

The *FORMAware* prototype is stable and completely implemented in Java. We have used it in some sample-cases (please refer to [10] which presents a broker-based digital library) and identified a few qualitative benefits, such as easy of use, expressiveness, extensiveness and integrity. More specifically, the generation of basic components is automatic (via *WrapperGenerator* tool) and the assembly of components is very intuitive (e.g. via *addComponent*, *plugInterfaces*, etc.) - please refer also to [9] for a simple description of the *FORMAware* usage. Furthermore, the structure of the assembly code is very simple thus suitable for graphic manipulation and automatic generation, diminishing the cognitive efforts for development. In addition, it was relatively easy to extend the default style manager to implement the *Broker* style (e.g. specialize the types of components, add stub/skeleton dependencies, etc.). Finally, the deployment of local and distributed components, the style enforcement and the transaction management are performed transparently, therefore making the development and adaptation of architectures simpler and more secure. Currently, we are extending the digital library to support *session continuity* and *service evolution* scenarios. The former *state-full* scenario undertakes dynamic reconfiguration of interface and communication components for adapting to user mobility, while the latter *state-less* scenario enhances the original digital library architecture with additional security and accounting/billing components. Despite being flexibility our major concern, we also consider fundamental to conduct an ample quantitative evaluation for measuring the performance costs directly associated with the interface indirection (i.e. proxy interfaces), constraints checking (i.e. style rules checking) and overheads of transaction policies. In addition, for comparison (local versus distributed) and extrapolation reasons (combination of architecture operations), we plan to measure the time needed to complete some atomic architecture operations (e.g. run, plug and remove components).

The ultimate goal of our approach is the ability to manage and constrain the run-time adaptability of software systems, i.e. perform safe architecture reconfigurations without breaking the integrity and functionality of applications. This is particularly important when we think about the plethora of pervasive systems and their need to adapt for coping with mobility, user interface variability and resource availability [15]. A different, though equally appealing, area of future work is related with self-healing or self-repairing systems that need to upgrade and repair bits of their architectures without stopping or rebooting the overall structure while preserving the architecture soundness. These areas of application constitute, we feel, an interesting environment for applying, extending and improving the principles enlisted and prototyped in *FORMAware*, i.e. capture the architecture semantic and bring it into operation for constraining and safely conduce architecture adaptation processes.


### Acknowledgments

Rui Silva Moreira is supported by FCT (Portugal) and *programme POCTI* (III European Union Communitary Supporting Framework) - grant SFRH/BD/4590/2001. Until September 2001, Rui was partially supported by the *University Fernando Pessoa (UFP).*


### References


[1] Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R. S., Parlavantzas, N., Saikoski, K., “The Design and Implementation of Open ORB V2”, IEEE DSOnline, 2001.

[2] Dowling, J., Cahil, V., The K-Component Architecture Meta-Model for Self-Adaptive Software, Reflection 2001: 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, Sept. 2001.

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *“Design Patterns: Elements of Reusable Object-Oriented Software”*, Addison-Wesley, 1994.

[4] Garcia-Molina, H., Ullman, J., Widom, J., *”Database System Implementation”*, Prentice Hall, New Jersey, 2000.

[5] Kiczales, G., J. des Rivières, D.G. Bobrow, The Art of the Metaobject Protocol , MIT Press, 1991.

[6] Kon, F., Supporting Automatic Configuration of Component-Based Distributed Systems, Proc. 5th USENIX COOTS99, 1999.

[7] Lavender, R., Schmidt, D., Active Object: An Object Behavioural Pattern for Concurrent Programming, "*Pattern Languages of Program Design 2*", Eds. Vlissides, Coplien and Kerth, AddisonWesley, 1996.

[8] Oreizy, P., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A., An architecture-based approach to self-adaptive software, IEEE Intelligent Systems, May-June 1999.

[9] Moreira, R., Blair, G., Carrapatoso, E., A Reflective Component-based & Architecture Aware Framework to Manage Architecture Composition, 3rd Int. Symposium on Distributed Objects & Applications, Roma, 2001.

[10] Moreira, R., Blair, G., Carrapatoso, E., *FORMAware*: Framework of Reflective Components for Managing Architecture Adaptation, 3rd Int. Workshop on Software Engineering and Middleware, Orlando, May 2002.

[11] TSE, *Software Architecture*, IEEE Transactions on Software Engineering, April 1995.

[12] Wang, N., Parameswaran, K, Schmidt, D., Othman, O., Evaluating Meta-Programming Mechanisms for ORB Middleware, IEEE Communications Magazine, pp102-112, Vol. 39, No. 10, Oct. 2001.

[13] Warren, I., "*A Model for Dynamic Configuration which Preserves Application Integrity*", PhD thesis, Lancaster University, 2000.

[14] Zarras, A., "*Configuration Systématique Synthesis de Middleware*", PhD thesis, Université de Rennes 1, 2000.

[15] Duran-Limon, H., "*A resource management framework for reflective multimedia middleware*", PhD thesis, Lancaster Univ., 2002.

[16] Costa, F., "Combining meta-information management and reflection in an architecture for configurable and reconfigurable middleware", PhD thesis, Lancaster University, 2001.