Supporting Adaptable Distributed Systems with *FORMAware*

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

The interactive and ubiquitous nature of future distributed services (e.g. digital libraries, learning systems, etc.) will make them more architectural and resource demanding. Consequently, next generation middleware frameworks should support both *shorter* and *longer-term* adaptation (i.e. *a priori* and *a posteriori* reconfiguration). In fact, current component standards [18] leverage software reusability and diminish development costs. Nevertheless, it is also a fact that existing middleware, which succeeded in meeting the goals of heterogeneity and interoperability, is not open enough for tackling the problem of dynamic evolution. This paper presents *FORMAware*, a framework that combines a novel component-based programming model enhanced by a reflective design, the former capturing the knowledge about software architecture abstractions (e.g. components, connectors, style managers, style rules) while the latter offering architecture awareness by explicitly opening the content and structure of both atomic and composite components (via introspection and adaptation meta-objects). In addition, *FORMAware* provides a set of tools and services for automating software development and adaptation (i.e. support for the generation, assembly, deployment and dynamic reconfiguration processes).

1. Introduction

We recognise the advantages of component-based development and the usefulness of component frameworks [16, 18]. We also understand that future distributed services will be more architectural and resource demanding. Consequently, next generation middleware frameworks should support reconfiguration for adapting to both *shorter-term* (e.g. upgrade components, scale to support more clients/servers, adapt to low bandwidth links, manage CPU and RAM restrictions or even user interface characteristics, etc.) and *longer-term* requirements (i.e. architecture evolutions for supporting extended services, security enhancements, new multimedia standards, etc.). Therefore, we propose a principled and systematic approach to adaptation for coping with a broader range of applications, user mobility, interaction devices and networks.

Current middleware solutions do not provide any architecture awareness and allow only design-time reconfiguration facilities (e.g. smart proxies, interceptors, containers) [12]. These reflective mechanisms introduce a degree of flexibility for choosing different management strategies and for adapting middleware behaviour to specific contextual applications. However, these approaches only allow static and predefined reconfigurations (e.g. transaction and security policies declared at design time) and are not generic and open enough for coping with broader adaptation scenarios (e.g. dynamic policy switching). More relevant (in terms of dynamic adaptation), is the lack of middleware abstractions for explicitly representing distributed composites, connectors and architectural constraints. In addition, the available meta-information facilities are currently limited to component types and local component dependencies that only address per-component evolution.

This paper proposes a reflective component framework called *FORMAware* for programmatically managing the adaptation of distributed systems. This framework opens the architecture semantic of composite components (i.e. architecture style [11]) and imposes explicit architecture constraints for observing and managing the architecture soundness. The combination of reflection and architecture awareness provides the mechanisms for safely managing the reconfiguration of distributed systems in order to cope with predictable or

even unpredictable adaptation requirements [9]. Some studies (cf. dynamicTAO [6] and K-Components [2]) address this important issue of safe/constrained adaptation by explicitly representing component dependencies. Nevertheless, none of the solutions recognises the need for an explicit style manager and associated set of style rules. In addition, the component dependencies are usually limited to local/direct bindings. Furthermore, some of the major issues behind adaptation (e.g. transference of state between replaced components, synchronisation of the reconfiguration process with the normal execution of applications) [13] are neglected or briefly considered. Our position proposes to extend architectural reflection for capturing domain specific semantics and use it for safely governing architecture adaptations. This paper follows with section 2 that presents the architecture details of *FORMAware*. Then, section 3 provides an application example and section 4 presents some evaluation observations. The paper finishes with a brief conclusion on section 5.

2. *FORMAware*, a platform for adaptability

2.1. Background technologies

In the last decade several research projects have investigated the use of *Architecture Description Languages* (ADLs) for describing, analyzing and evaluating systems architectures [8, 11, 14]. However, current ADL frameworks do not incorporate the architecture semantic (e.g. architecture manager, style constraints) at the programming level. 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 [10]. We argue that the architecture semantic should be represented in the programming model for promoting the proximity between design and development and, therefore, allowing us to combine the software architecture best practices [11] for managing run-time reconfiguration. Moreover, we believe that combining *design patterns* and *reflection* permits us to capture the design knowledge and safely expose it for increasing the levels of flexibility/adaptability. None of the current ADL-frameworks provides such an open and integrated architecture awareness approach. In fact, we propose a component-based programming model extended *with* reflective meta-objects for exposing the content and topology of both basic and composite components (see [9, 10, 17] for more details).

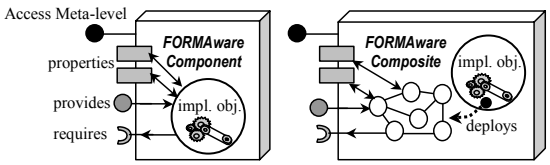

Figure 1: Structure of *FORMAware* components.

2.2. Structuring the meta-level

2.2.1. Overview

FORMAware defines a root component (*RComponent*) with basic characteristics inherited by both basic and composite components (see figure 1). This class provides access to introspect and adapt meta-objects (cf. *method factory* design pattern [3]) which implement specific *Meta-Object Protocols* (MOPs) for examining or modifying the details of each architecture element (cf. *interface* and *architecture meta-models* [1]). More specifically, the design of the framework addresses structural reflection, which is concerned with the content and topology of components (e.g. access the provided and required interfaces of components, set/get their properties, browse the graphs of composites, check their architecture manager and style rules, etc.). More interesting is the *architecture meta-model* which permits us to explicitly assemble components (*awareness of composition*), choose the type of topology (*awareness of architecture style*) and select the rules governing it (*awareness of constraints*), as explained below.

2.2.2. Awareness of composition

Composite components (*RComposite* class) represent configurations of architecture elements (i.e. components and connectors) plugged together through provided and required interfaces. These composites can export properties and interfaces, provided or required by their internal components. In addition, each composite has an implementation object (similar to basic components) that is responsible for the configuration algorithm i.e., creating, plugging and deploying the internal components. The topology of each composite is shaped by a specific architecture strategy as described next.

2.2.3. Awareness of architecture style

The composition model proposed by *FORMAware* is not flat since it uses a *style manager* responsible for governing an explicit *architecture* graph (*context*) of component (see figure 2). This design (cf. *strategy* design pattern [3]) permits us to guarantee the architectural soundness of each graph reconfiguration. *FORMAware* defines a *default architecture style* which

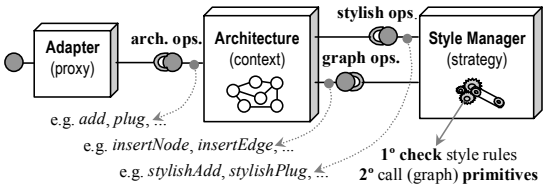

Figure 2: structure of the architecture meta-model.

may be extended, by developers, to conform to other specific architecture requirements (e.g. *Layer*, *PipeFilter*, *Broker*, *Blackboard*).

Developers use adapter meta-objects for performing architecture operations on composite components (e.g. add and plug components, export or import interfaces, establish dependencies between an *encrypt* and counterpart *decrypt* component, etc.). Each operation invoked on the adapter is passed to an equivalent *architecture operation* available in the architecture context. Then, the architecture forwards it to an associated *stylish operation* provided by the style manager (see figure 2). The *stylish* operation executes a series of style *checks* before calling-back the *primitive* graph-operation on the architecture graph. The style verifications follow a certain *template*, according to the type of architecture operation and groups of rules used by the style manager (cf. *template method* design pattern [3]). For example, for binding 2 components the *stylishPlug* operation, 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 then execute the *primitive plug* method that calls-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*) enables the customisation of the 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

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]. Developers may extend the default set of constraints with new rules enforcing additional checks.

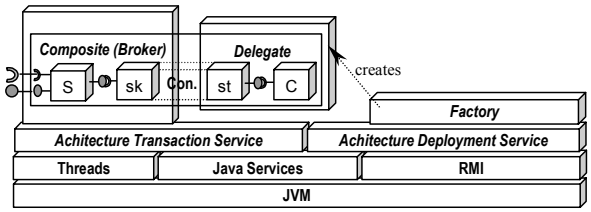

Figure 3: architecture of FORMAware.

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. Managing distributed adaptation

2.3.1. Overview

FORMAware is entirely implemented in Java and provides mechanisms for deploying components on remote machines and also for transparently and safely managing architecture reconfigurations. This is supported by 2 services: deployment and adaptation transaction services (see figure 3).

2.3.1. Architecture deployment service

Deployment relies on the notion of *delegate* components which are ambassadors of composites running on remote machines. Each composite uses a factory for creating delegates which then allow them to execute visitor operations (*visitor* design pattern [3]) to remotely run, kill, plug and unplug components. The deployment service uses RMI for distribution purposes; hence, visitor objects are passed by value but need to be serializable. However, the connectors used in the applications may implement other forms of communication (e.g. CORBA, publish-subscribe, etc.).

This design ensures a thin delegate component, which only needs to execute a generic *callback* on each visitor operation. In addition, the set of visitor operations may easily be extended without changing the delegate. This way, composites remain responsible for selecting and checking style rules and delegates behave like slaves that execute the master requests. This approach is also safer since only the composite can send visitor operations to be executed on the delegate.

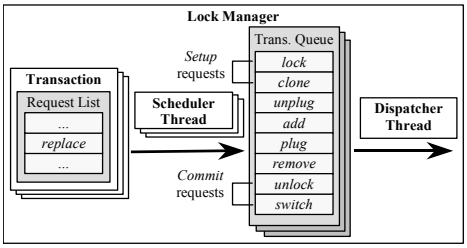

Figure 4: transaction service - requests for a "*replace*".

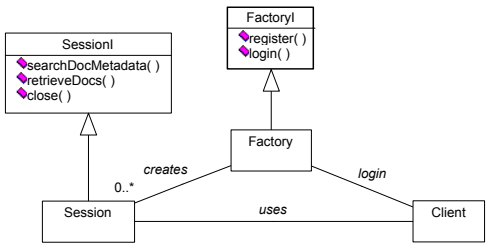

Figure 5: UML design of the monolithic digital library service.

2.3.2. Architecture transaction service

The architecture style manager guarantees the consistency of the configurations but does not synchronise the process of reconfiguration [10]. For that we provide a transaction service to initiate, commit and rollback architecture operations without compromising the application execution. The design of the service combines a basic scheduling architecture (cf. *locking scheduler* [4]) with a behavioural pattern (cf. *active object* design pattern [7]) for managing the concurrent nature of the adaptation process. The service possesses a *transaction manager* that creates architecture *transactions* (on-demand) and *enlists* them on the *lock manager*. For each enlisted transaction, the lock manager creates a *scheduler thread* that is responsible to *get* the transaction requests, interpret them and, then, *generate/enqueue* specific intermediate requests in the request stream of the lock manager. For example, figure 4 shows that the scheduler *enqueues* the *lock*, *clone*, *add*, *unplug*, *plug*, *remove*, *unlock* and *switch* requests for a *replace*. These requests depend on the selected scheduler policy. 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 from all schedulers.

The transaction service allows developers to choose the adaptation policy (e.g. *SKIP, WAIT or FORCE* safe state). Each scheduler uses specific *setup* and *commit* action requests. The *SchedulerSkipSafeState*, for example, *enqueues* the *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. Similarly, the *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). At *commit-time* each scheduler *enqueues* final requests to *transfer* the state and *switch* the old components by the new ones.

3. Creating a Digital Library Service

3.1. Baseline Digital Library

In [15] we have presented a prototype digital library for accessing a repository of digital documents, which was developed with COTS frameworks (see baseline design in figure 5). The service required a local network access, an SMTP-based registration and download functionality, and a simple search mechanism by title and author. This service was completely functional but rather monolithic and with few adaptation capacities, i.e., not open enough to adapt to new requirements (e.g. upgrade to an advanced search functionality, reconfigure GUI and communication components to support mobile users, etc.).

3.2. Componentization

FORMAware permits us to re-work this service into a specific style-conformant and adaptable component-based architecture. More precisely, the *Session* and *Factory* classes are used for generating basic components. In addition, we separate these classes from the communication and GUI concerns. Consequently, we develop new components for sending SMTP messages (*Mail*), establishing JDBC connections (*DBUsers* – check permissions of users; *DBDocs* - search/download documents) and provide specific client-side GUIs (*LoginGui* and *SearchGui*). We also detach the communication issues from the components by using separate RMI-based connectors. Notice that components are automatically generated from the respective implementation classes with the help of the *WrapperGenerator* tool (see [9] for details).

3.3. Specialize the *Broker* style

The default style manager does not enforce the relationship between the *stub* and *skeleton* roles of a RMI connector. Moreover, it does not guarantee the cardinality of interfaces provided by each RMI connector (i.e. existence of a specific interface that provides the connectivity and also a generic *TieRmiRI* interface for tying the stub with the skeleton). In

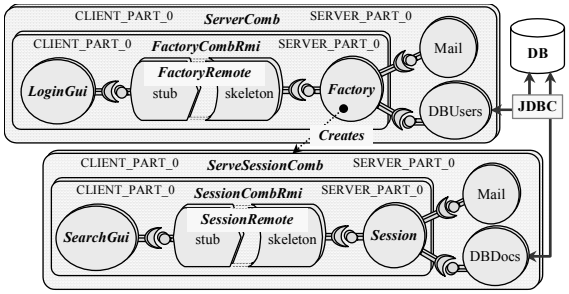

Figure 6: configuration of the digital library service.

addition, for broker style composites we may specialise the types of components allowed (e.g. *Client*, *Server*, *RmiConnector*) thus enriching the semantic of the architecture style. For these reasons we have created the *Broker* style that simply extends the *DefaultArchitectureStyle* by changing some of the default style rules and also incorporating new rules.

3.4. *A priori* (re)configuration

We can now use the Broker style for (re)structuring the digital library service, assembling the components into different self-contained composites (see figure 6). The composites are instantiated and plugged in parts and may extend over several roles (e.g. *SERVER* and *CLIENT*). For example, the *factory-related* components are instantiated, plugged and deployed independently from the *session-related* components. In particular, the *Factory* component is responsible for triggering the deployment of the *ServeSessionComb* composite. Therefore, each role/part is deployed by a specific Java-script that executes the necessary architecture operations (e.g. create, add, plug components, etc.). The deployment scripts are named according to the *role* and *part* that need to be deployed (e.g. the script *ServerComb_SERVER_PART_0* creates the *Mail* and *Users* basic components and then triggers the script for deploying the *PART_0* of the *FactoryCombRmi* composite on the *SERVER* side).

3.5. *A posteriori* reconfiguration

Imagine now that we need to extend the digital library architecture to allow *advanced search* (i.e. by author, title, abstract and date) and support *session-continuity* (i.e. allow to *pause* a session and *resume* it afterwards). Moreover, we want to provide access through a PDA with limited connection and GUI facilities (see figure 7). For supporting these requirements we develop new components (e.g. *PalmFilter* - parses http requests and formats the response according to PDA GUI capabilities) and extend some existing components (e.g. *FactoryX*, *SessionX, UsersX*) and connectors (e.g. *SessionXRemote*) for incorporating the new

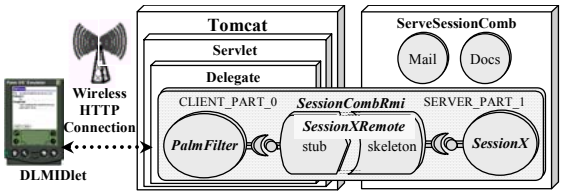

Figure 7: initial digital library service.

functionalities. Then, we create/update the scripts for deploying the new session-related components. Finally, we use the reflective and transaction mechanisms provided by *FORMAware* (via the adapter meta-object of the *ServerComb* composite) for executing a reconfiguration script that dynamically replaces the *factory-related* components (*SERVER* role) without stopping the service. Please refer to [9, 10] for some code samples showing both architecture deployment and reconfiguration.

4. Evaluation

4.1. Quantitative

FORMAware introduces composition awareness through provided and required interfaces. This design, however, carries an indirection overhead since these interfaces introduce *pre* and *post* processing actions (for handling state information and testing/throwing *Blocked* and *Unplugged* exceptions). Figure 8 presents the average time taken by a 10^7 *for-cycle* executing a method call invoked: via a interface plug (i.e. a required interface plugged on a provided), via a provided interface and directly on a Java object.

The results show that the overhead is not high when we compare the average time taken by the Java calls in contrast with the equivalent method calls performed on the interfaces without any pre/post processing i.e., 1,56 times more for calls via the provided interface and 2,23 times more for calls via the interface plug. The overheads are slightly higher when we introduce the pre/post exception handling, respectively 2,35 times higher for calls via the provided interface and 3,16 times higher for calls via the interface plug. However, the overhead increases significantly when we consider both the pre/post state and exceptions processing (respectively 6,04 and 6,89 times more than the equivalent Java calls). We can also note that the pre/post state processing facilities introduce most of the overhead. Nevertheless, developers may decide to handle state management themselves (by implementing the *ManageStateI* interface) and deactivate the generation of the pre/post state actions and thereby controlling/diminishing the overhead.

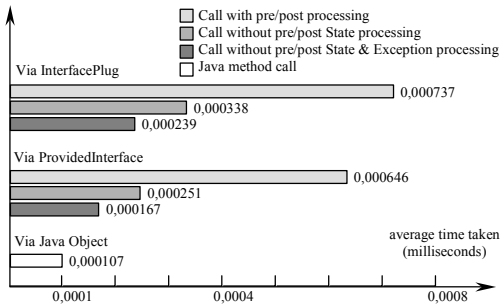

Figure 6: overhead on method calls.

4.1. Qualitative

The *FORMAware* prototype is stable and was used for developing some sample-cases (including the presented digital library service). This permitted us to corroborate a few qualitative benefits, such as extensiveness, easy of use, expressiveness and integrity. More specifically, the extension of the framework itself (i.e. creating the *Broker* style) was a straightforward process that allowed us to improve the architecture semantic for increasing the architecture soundness. Moreover, the generation of basic components is automatic (using the *WrapperGenerator* tool) and the assembly of components is very intuitive (e.g. via *addComponent*, *plugInterfaces*, etc.) thus providing a natural mechanism for programming the deployment of applications. 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, 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. Finally, the presented evolution scenario demonstrates that it is relatively easy and architecturally safe to extend the digital library for supporting *session continuity* i.e., undertake dynamic reconfiguration for adapting to user mobility.

5. Conclusion

This paper purposes a framework for managing and constraining the run-time adaptation of distributed systems, i.e., performing safe architecture reconfigurations without breaking the integrity and functionality of distributed 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. 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, a challenging environment for applying, extending and improving the principles enlisted and prototyped in *FORMAware*, i.e., capturing the architecture semantic and bring it into operation for constraining and safely conduce architecture adaptation processes.

Acknowledgments

Rui Silva Moreira is supported by FCT and *programme POCTI* (III EU Communitary Supporting Framework). Until September 2001, Rui was partially supported by the *UFP*.

References

[1] Blair, G. et al., "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. et al., *"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. et al., 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 DOA, Roma, 2001.

[10] Moreira, R., Blair, G., Carrapatoso, E., *FORMAware*: Framework of Reflective Components for Managing Architecture Adaptation, 3rd Int. Workshop SEM, 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, pp102-112, Vol. 39, No. 10, Oct. 2001.

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

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

[15] Carrapatoso, E., Moreira, R., Oliveira, J., Mendes, E., *Development of a Distributed Digital Library: from Specification to Code Generation*, JECT'99, Lisboa, October 1999.

[16] Fayad, M. E., Schmidt, D. C., Object-Oriented Application Frameworks, Communications ACM, Vol. 40, No. 10, October 1997.

[17] Moreira, R., Blair, G., Carrapatoso, E., Constraining Architectural Reflection for Safely Managing Adaptation, in Proc. 2nd WS on RAMS (Middleware03), pp139-143, Rio Janeiro, BR, 2003.

[18] Emmerich, W., "*Engineering Distributed Objects*", John Wiley & Sons, Chichester, UK, 2000.