The K-Component Architecture Meta-Model for Self-Adaptive Software

Jim Dowling and Vinny Cahill

Distributed Systems Group
Department of Computer Science
Trinity College Dublin
Jim.Dowling@cs.tcd.ie, Vinny.Cahill@cs.tcd.ie

Abstract. Software architectures have recently emerged as a level of design concerned with specifying the overall structure of a system. Traditionally, software architectures only provide static descriptions of the participants and interaction structures in a system. Dynamic software architectures, however, can be reconfigured at runtime and therefore provide support for building dynamically adaptable applications. Software architectures can be specified using architectural reflection. In this paper we introduce an architecture meta-model that realises a dynamic software architecture. The architecture meta-model reifies the configuration graph of the architecture and is automatically generated from our component definitions and implementation language source-code. We show how graph transformations that re-write the architecture's configuration graph can be implemented as reflective programs, called adaptation contracts. Adaptation contracts are written in a separate programming language, thus cleanly separating the adaptation code from the computational code. Adaptation contracts can even be replaced at run-time. They are deployed in a run-time meta-level architecture that addresses issues of system safety, integrity and overhead during graph transformation. The paper also describes a prototype implementation of our model called K-Components.

Keywords: Dynamic Software Architectures, Architectural Reflection.

1 Introduction

Architectural structures have always been present in object-oriented software, but their importance has increased with the advent of complex, distributed object systems. Modelling complex systems can be done informally with system diagrams or more formally using an Architecture Description Language (ADL) [SG96]. Traditionally ADLs were developed for representing and reasoning about the static structures and interactions in a software architecture [AdlSei]. However, since there are many classes of systems that are dynamic, including evolving systems that require on-line software upgrades, adaptive systems whose provided behaviour adapts to resource availability in their current operating environment, and learning or knowledge-based systems, there has recently emerged a field of study concerned with dynamic software architectures.

Dynamic software architectures can be used to build dynamic systems by supporting the self-management and reconfiguration of the dynamic system's architecture at run-time [Allen98]. Current approaches to specifying dynamic software architectures use ADLs with Architecture Modification Languages [Darwin95, OGT99, Rapide95, Werm00] or Co-ordination Languages [Cuesta01]. Another approach to specifying dynamic software architectures is to use *architectural reflection* [Caz00]. A system that supports architectural reflection reifies its architectural features as an *architecture meta-model* [Blair01] that can be inspected and modified at run-time. Modifications of the architecture meta-model result in modifications of the software architecture itself, and the architecture is therefore reflective. The architecture meta-model is not only concerned with the architectural features it reifies but also with an associated set of *architectural constraints* [Blair01], describing how and when to safely reconfigure the software architecture.

The architecture meta-model introduced in this paper realises a dynamic software architecture. Our architecture meta-model reifies the software architecture as a typed, directed configuration graph, where interfaces are the vertices, components the type labels and connectors are directed edges. We present a novel approach to generating the architecture meta-model. Rather than force programmers to explicitly specify the software architecture's configuration graph with an ADL, we automatically generate the meta-level configuration graph by building a dependency graph from both the component definitions, in Interface Definition Language-3 (IDL-3) [CCM99], and the connections between them, defined in the system implementation language.

The architectural constraints over the architecture meta-model enforce the dynamic properties of or assertions about the software architecture and are encapsulated in reflective programs called *adaptation contracts*. We use the more general term *adaptation code* to mean the architectural constraints of a system. To obtain a clean separation of concerns between the adaptation code and computational code at design time, we provide a separate *adaptation contract description language* for specifying the adaptation code as adaptation contracts.

Adaptation contracts can reason about the system's architecture, state and external dependencies using adaptation events and reconfigure the system's architecture and external dependencies using reconfiguration operations. Since adaptation contracts invoke reconfiguration operations on the architecture meta-model that result in modifications of the software architecture itself, adaptation contracts are reflective programs. Adaptation contracts are separate meta-level objects and they can be loaded and unloaded at run-time. A container, called a configuration manager, provides a deployment and execution environment for the adaptation contracts and also allows for the injection of new adaptation contracts at runtime.

In summary, we model dynamic reconfiguration as the transformation of a system's architecture meta-model and use adaptation contracts to separate adaptation code from computational code.

2 Background and Related Work

Dynamic systems are systems that can adapt their behaviour while executing, relative to some context information. Dynamic systems come in two flavours: *closed dynamic systems* and *open dynamic systems* [OGT99]. For closed dynamic systems both the complete system behaviour and the behaviour that describes adaptations are specified at build time. Open dynamic systems, on the other hand, allow system behaviour to evolve after build time and are necessary if unanticipated adaptations are to be performed at run-time. Techniques for implementing open dynamic systems include dynamic linking mechanisms, dynamic object technology (including class loaders) and dynamic programming languages [OGT99].

It is also important to make the distinction between adaptable and self-adaptive systems. *Adaptable* systems can be adapted to a particular deployment environment [Czarn00], whereas *self-adaptive* systems adapt themselves to their operating environment [OGT99]. Adaptable systems support their *explicit adaptation* by an external actor [Werm00] using either a procedural or declarative interface [Blair01]. Adaptable systems with procedural interfaces are open dynamic systems and can support arbitrary or evolutionary adaptations [MG99].

Self-adaptive systems, however, are subject to *implicit adaptation*, triggered by changes in either their internal state or their environment. Self-adaptive systems possess adaptation logic. *Adaptation logic* is the code that monitors a representation of the system's internal state and environment and then adapts that representation resulting in a reconfiguration of the actual system. Adaptation logic is by nature reflective. Self-adaptive systems are normally closed dynamic systems. A closed self-adaptive system is said to support programmed adaptation [MG99]. If a self-adaptive system is to be open dynamic, it has to support the dynamic loading of adaptation logic. Self-adaptive systems require no support from external actors, and so can be adapted transparently to users of the system. Implicit adaptation is required to achieve *adaptation transparency*. Application-transparent adaptation is desirable when the conditions for adaptation can be known a-priori. Fig.1 shows an architecture that supports both its explicit and implicit adaptation.

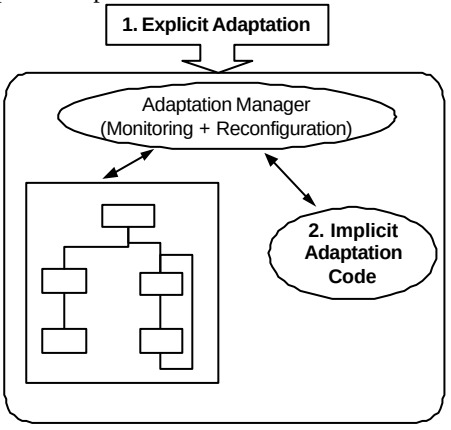

Fig. 1. Explicit and Implicit Adaptation with an Adaptation Manager

2.1 Adaptation Logic

Adaptation logic for a system is usually made up of two main parts – *monitoring* and *reconfiguration* code, as can be seen in [Silva00, QuO00]. Monitoring code is responsible for identifying operating region-transitions [QuO00] in a system. Operating regions define the different operating modes of a system and each one generally requires a different configuration of the system. Operating region transitions can be triggered either by some internal state change in the system or by a change in a system that it is dependent on [Kon01]. Operating region transitions identify adaptation events, events that signal the need for a reconfiguration of the system. For example, when resource managers can no longer provide their managed resource to clients at some quality of service level they will generate adaptation events that are both handled internally and sent to resource-aware client systems. Systems can generate and receive adaptation events, handle them internally and forward them to interested clients.

While adaptable systems possess only monitoring code and an interface for invoking reconfiguration commands, self-adaptive systems contain both monitoring and reconfiguration code. Thus, the intelligence for when and how to reconfigure the system is either supplied externally by a user and is, therefore, modifiable at run-time (i.e. adaptable systems), or else it is supplied internally and known at design time (i.e. self-adaptive systems). Systems can be both adaptable and self-adaptive.

The run-time infrastructure that manages the monitoring and reconfiguration aspects of the system can be either centralised (as in an adaptation manager, see Fig.1) or distributed (as in self-organising architectures) [MG99]. Variations of the adaptation manager for self-adaptive systems can be found in the configuration manager [KM98, Werm00] and in the configurer [Allen98, Kon01].

2.2 Separating Adaptation Code from Computational Code

To obtain a clean separation of concerns between the adaptation code and the computational code at design time, a separate language can be used to specify the adaptation logic. Examples of this include the configuration programming philosophy [KM98] that advocates using two languages for configuration programming: a configuration language for the structural description and change logic and a separate programming language for basic component programming. Similarly, BBN's Quality Objects uses separate Quality Description Aspect Languages [QuO99] to write a distributed object's quality of service contracts. Techniques such as aspect-oriented programming [Czarn00] and reflection can enable the use of separate languages to represent separate concerns in a program.

2.3 Dynamic Software Architectures

Dynamic software architectures represent a principled method for the construction of adaptable and self-adaptive systems. Software architectures represent static metadata about the components, the connectors and the architectural configuration of a system and they are often specified explicitly using an Architecture Definition Language (ADL) [SG96]. To support dynamic software architectures, there is a need to support adaptation logic and provide mechanisms for managing the system's integrity and dependencies during reconfiguration. There is, however, little consensus in the research community on what architectural features a dynamic software architecture should contain. Different approaches to the specification and construction of dynamic software architectures include Event Systems [Rapide95, OGT99], process algebras in Wright [Allen98], graph grammars [LeMet98], rewriting logic [Werm00] and architecture meta-models [Caz00, Blair01, Cuesta01].

Adaptation logic in software architectures is captured in both architectural constraints and reconfiguration operations. Architectural constraints are properties of or assertions about configurations, component or connectors, the violation of which will render the software architecture unacceptable (or less desirable) to one or more stakeholders [MT00]. Examples of reconfiguration operations include the addition and removal of components and their linking and unlinking via connectors [Werm00].

In the case of mobile systems, a good deal of the adaptation logic is known in advance, indicating that they would benefit from application-transparent adaptation. But in order to adapt to unforeseen and exceptional system states, there is also a requirement for dynamic updating of their adaptation logic. Currently no dynamic software architecture provides support for the dynamic loading and unloading of programs with adaptation logic (i.e. as architectural constraints) and we propose the use of adaptation contracts for this purpose.

2.4 Architectural Reflection

We define architectural reflection as being concerned with the observation and manipulation of the configuration graph of a software architecture and its constituent vertices and edges. Just as there is no consensus on what architectural features should be in an ADL, there is no consensus on what constitutes the vertices and edges in a software architecture's configuration graph. For example, the configuration graph's vertices and edges are components and connectors in [MOT00], interfaces and connectors in [Rapide95] and interfaces (labelled with an implementation component) and connectors in [Werm00], representing a typed configuration graph. Behavioural reflection is concerned with the reification of a system's computation. In software architectures, each configuration graph represents a particular computational instance of the system. We define behavioural reflection for software architectures as the ability to rewrite the system's configuration graph of components and connectors at runtime. Structural reflection for software architectures is concerned with introspecting the architecture's configuration graph and constituent components, connectors and interfaces.

3 Architecture Meta-Model

An architecture meta-model reifies the architectural features of a system. The main issues in designing an architecture meta-model for a dynamic software architecture include what architectural features to represent in the meta-level (e.g. the architecture's configuration graph, components and connectors), how to represent the adaptation code, what mechanisms to provide for integrating the adaptation code into the system, and how to manage system integrity and consistency during dynamic reconfiguration. The following sections introduce the features in our architecture meta-model.

3.1 Architecture Meta-Model's Configuration Graph

We reify a software architecture configuration as a typed, connected graph, where the vertices are interfaces, labelled with component instances, and the edges are connectors, labelled with connector properties. A vertex is modelled as an interface and implementation (component) pair, *(i,c)*. An edge is modelled as a triple $i \rightarrow_l j$, which contains the source and target vertices ids *i* and *j*, and the edge label *l*. The edge label represents reconfigurable properties of the connector such as the ability to change its communication protocol or set of installed interceptors. The root vertex of a configuration graph is a special type of vertex, the entry point in the program. It is normally the main() of a C++/Java implementation. Cycles are allowed in the graph and are modelled with cyclic connectors. A meta-level component, called the configuration manager, is responsible for the storage and management of the software architecture's configuration graph.

3.2 Dynamic Reconfiguration as Configuration Graph Transformation

We represent dynamic reconfiguration as conditional graph transformations, specified in reflective programs called adaptation contracts. A graph transformation is a rule-based manipulation of the configuration graph [Werm00]. The interfaces and connectors that represent the vertices and edges in our graph describe the static structural part of the system that is preserved during a graph transformation. The component instances and connector properties that represent the labels of the vertices and edges in our graph respectively describe the dynamic structural part of the graph that is rewritten during a graph transformation.

Since graph transformations ensure that the result of a rule is again a graph, we can guarantee the integrity and consistency of the system if the graph rules are *transactional operations* over the graph. In practice, however, graph transformations may affect only part of the configuration graph and a *reconfiguration protocol* [KM98] can be used to ensure that only those vertices that are affected by the transformation must be in a *safe state* [Werm00]. We follow this approach and use a reconfiguration protocol as it helps reduce the length of the reconfiguration phase and allows concurrent client invocation during the reconfiguration phase on components that are not "frozen". Computation and adaptation code are related through the reconfiguration protocol as it *freezes* computation in components involved in a reconfiguration. Component state can only be changed by computation, not by reconfiguration operations. One of the other advantages of our reconfiguration protocol is the maintenance of system state integrity by transferring component state from the old component to the

new one. The successful transfer of component state requires that component developers implement a copy constructor interface for their component. The meta-level configuration manager is responsible for implementation of the reconfiguration operations and the correct operation of the reconfiguration protocol.

3.3 Managing Dynamic Reconfiguration

The configuration manager [KM98, Werm00] is a meta-level adaptation manager that stores the reified configuration graph and implements the reconfiguration (graph rewrite) operations and reconfiguration protocol. The configuration manager is a run-time container for the deployment, scheduling and execution of adaptation contracts, see Fig. 2. The configuration manager provides a procedural interface for the loading/unloading of adaptation contracts at runtime. In order to guarantee the safety of the code being introduced into the system, the new adaptation contracts must be developed and tested in the original, trusted development environment. The procedural interface verifies that the new adaptation contract is trusted from its XML descriptor.

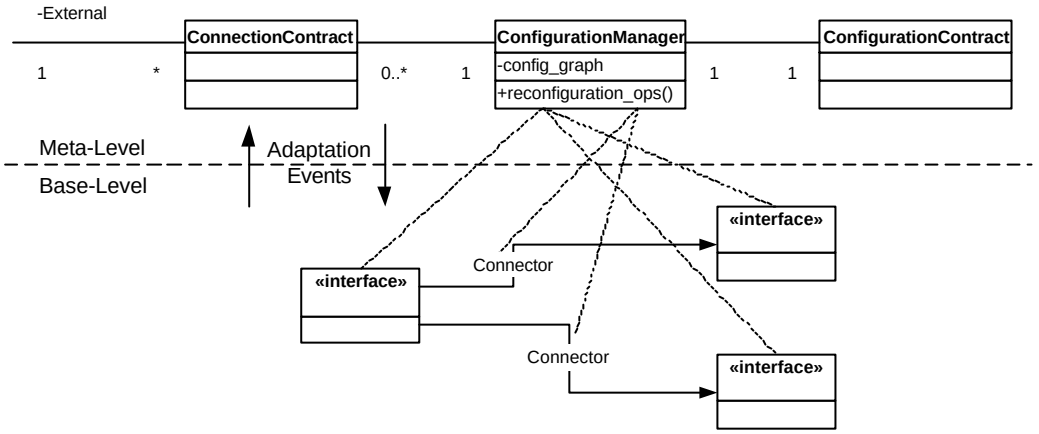

Fig. 2. Configuration Manager as a Meta-Level Adaptation Manager

4 Adaptation Contracts

Adaptation contracts are used to specify meaningful transformations of an architecture meta-model's configuration graph. They can be used to provide the adaptation logic for self-adaptive systems. Adaptation contracts are reflective as they invoke reconfiguration operations on the architecture meta-model that results in modifications of the software architecture itself. They are represented at runtime by meta-level objects and are deployed in and managed by a configuration manager, see Fig. 2.

An adaptation contract contains a series of conditional rules for the transformation of the meta-level configuration graph. Adaptation contracts are used to specify a system's architectural constraints. Since architectural constraints represent properties of or assertions about configurations, components or connectors, our adaptation contracts require a mechanism for accessing these properties and assertions. We provide *adaptation events* as a mechanism for allowing contracts to poll architectural constraint information from meta-level configurations and base-level components and connectors. Architectural constraints can be specified using adaptation events with conditional statements.

Adaptation events also have the advantage of decoupling the meta-level adaptation contract from the base-level components and connectors, see Fig. 2. In effect, they provide a run-time separation of concerns between the adaptation code and the computation code. This allows adaptation contracts to be dynamically loaded and unloaded, since they have no clients directly dependent on them.

During the reconfiguration of a software architecture, the management of the *incoming* and *outgoing dependencies* of a system (the systems/users that depend on it and the systems that it depends on, respectively) is crucial to the safety and integrity of both the system and its dependent systems [Kon01, Dow00]. Because of the two different types of dependencies, adaptation contracts come in two forms: *configuration contracts* that manage internal resources and incoming dependencies from clients, and *connection contracts* that manage outgoing dependencies to components the system is dependent on.

```
/* Automatically generated from XML configuration descriptor */
configuration  configuration*          ::= configuration -descriptor.xml
interface      interface*              ::= component-descriptor.xml

/* Automatically generated from component XML descriptors */
component      component-name*         ::= component-descriptor.xml
consumes       user_defined_handler*   ::= component-descriptor.xml
emits          adaptation_event*       ::= component-descriptor.xml
handler        handler*                ::= reconfiguration_op | user_defined_handler
reconfiguration_op                     ::= change_component | change_configuration |
                                           install_interceptor | remove_interceptor | rollback |
                                           callback_client
/* User-defined below here*/
contract configuration configuration
{
      /*      Conditional statements and reconfiguration operations      */
}

contract connection connection-contract*        ::=
      [component client_name] [interface port_name] [component provider_name]
{
      /*      Conditional statements and reconfiguration operations      */
}
```

Table 1. Adaptation Contract Description Language Syntax

Configuration contracts are specified in the Adaptation Contract Description Language, see Table 1. They consist of a series of conditional statements, testing for the occurrence of adaptation events, and associated reconfiguration operations. There is support for managing incoming dependencies in the form of a contract renegotiation. A contract renegotiation informs dependent clients of a reconfiguration in the system. In our model, configuration contracts can only be defined for a single-address space. *Connection contracts* represent architectural constraints for individual connections between the system and components it is dependent on (potentially outside the address space of the system). Similar to configuration contracts, they consist of a series of conditional statements and reconfiguration operations. A *configuration tool* takes adaptation contracts, the components and connectors in the software architecture and produces an implementation in a concrete language, i.e. in our prototype C++. It performs refinement transformations by allowing the specialisation and concretisation of the abstract meta-level components in the K-Component framework.

5 The K-Component Framework

K-Components are components with an architecture meta-model and adaptation contracts to support their dynamic reconfiguration. Our prototype implementation is in C++. The following section describes the component and connector models, how the software architecture is automatically generated from the implementation programming language and how to attach the adaptation contracts to architectures.

5.1 The Component Model

We leverage existing component syntax to define components. IDL-3 [CCM99] is used to define components with explicit dependency management, through `provides` and `uses` interfaces. IDL-3 `emits` and `consumes` events [CCM99] are used to specify base-level adaptation events. Component skeletons are generated by the IDL-3 compiler by specialising and templating abstract components in the K-Component framework. Components can be either *primitive* or *composite* components [Darwin95]. Only composite K-Components have their own architecture meta-model, configuration contract and configuration manager.

Our C++ framework supports the *dynamic evolution* [MT00] of components using component subtyping, factory objects [COM+99] and dynamic linked libraries (DLLs). Components also have

interfaces that are used by the reconfiguration protocol: a traversal interface for traversing the configuration graph and a copy constructor for migrating connectors and state during component replacement. Reference counting is used to guarantee a safe-state during the reconfiguration phase and for the safe removal of components. To overcome the cyclic reference counting problem [COM+99], programmers explicitly specify connectors as being either normal or cyclic connectors.

We do not need full introspective meta-level information about components at runtime in order to rewrite the configuration graph, therefore we do not represent components as full meta-level objects. They are represented in the meta-level by a typed reference in the configuration manager, used for reconfiguration operations and adaptation event processing. Component meta-information, for use in the adaptation contract description language, is stored in an XML component descriptor [CCM99] that is generated from the component's IDL-3 definition. The programmer fills in additional details such as component packaging and deployment information.

5.2 Connectors as First-Class Entities

In K-Components, connectors are implemented as typed objects relating `provides` and `uses` interfaces on components. Connectors are generated from IDL-3 component definitions by specialising and templating abstract connectors in the K-Component framework. There are two types of connectors: *client-side* and *provider-side connectors*. Client-side connectors can connect directly to a component, unless it is external to the system in which case it connects to a provider-side connector on the target component. For correct functioning of the reconfiguration operations, programmers have to ensure that they explicitly represent cycles in their configuration graphs as cyclic connectors. Connectors provide a reconfiguration interface, with operations such as `link_component` and `unlink_component`, and the configuration manager uses this interface to implement its graph rewrite operations.

5.3 Writing and Configuring K-Components

C++ is used instead of an ADL to specify the software architecture. Several abstractions and programming idioms are used in our C++ prototype implementation for representing concepts commonly found in an ADL, such as interfaces, connectors and binding operations. In addition to this programming guidelines are specified, e.g. for services offered by an interface can only be accessed via connectors.

Rather than force programmers to explicitly specify the software architecture's configuration graph with an ADL, we automatically generate the static part of the meta-level configuration graph by building a dependency graph from the C++ source-code. Similar to how a C++ compiler generates a dependency graph from C++ header files, the configuration tool parses the C++ source code from its entry point for connectors (edges) and their target interfaces (vertices). Each interface's header file is then parsed in turn for more connectors and interfaces until leaf interfaces are reached and a full parse-tree of the system is built. The configuration tool then produces a typed, directed configuration graph of the system with interfaces as vertices and connectors as edges as an XML configuration descriptor.

The programmer can bind the interfaces to actual components by editing the interface labels in the configuration descriptor. Once component implementations have been specified for all the interfaces in the configuration graph and adaptation contracts have been attached, the software architecture can be instantiated by the configuration tool.

6 Conclusions

Without a mechanism for achieving a separation of concerns, dynamic systems' adaptation-specific code becomes tangled with its functional code. We provide an adaptation contract description language that separates the adaptive behaviour of systems from their functional behaviour. We are able to provide a separate language for the adaptation code since dynamic reconfiguration is implemented using reflective programs called adaptation contracts. Architectural reflection enables this separation of concerns.

Our architecture meta-model reifies the software architecture of a system as a typed, directed configuration graph with interfaces as vertices, labelled with component instances, and edges as connectors. We model dynamic reconfiguration as a transformation of a system's configuration graph, and use a reconfiguration protocol to guarantee the consistency and integrity of both the

reconfiguration operation and the system. Meta-level graph rewriting programs, called adaptation contracts, perform conditional graph transformations. Adaptation contracts allow programmers to write adaptation transparent applications.

Our model of dynamic reconfiguration is constrained to replacing the components in a system's configuration graph. The alternative of allowing new services to be introduced to a system at runtime leaves open the problem of how existing components in the system and existing clients of the system become aware of and access these new service interfaces at runtime. For self-adaptive software, we do not see our model of dynamic reconfiguration as being overly restrictive. In fact, it can help programmers by constraining the system's possible dynamic reconfigurations to meaningful ones.

Bibliography

[AdlSei] Carnegie Mellon Software Institute, *Architecture Description Languages*, URL = http://www.sei.cmu.edu/str/descriptions/adl_body.html.

[Allen98] Robert J. Allen, Remi Douence, and David Garlan, "Specifying and Analyzing Dynamic Software Architectures", *Conference on Fundamental Approaches to Software Engineering, March 1998.*

[Blair01] Gordon Blair et Al., "The Design and Implementation of Open ORB v2", DS Online Vol. 2, No. 6 2001.

[Caz00] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato, "Explicit Architecture and Architectural Reflection". In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects* (EDO 2000), LNCS. Springer-Verlag.

[CCM99] OMG, *The CORBA Component Model*, orbos/99-07-01.

[COM+99] Guy and Henry Eddon, *Inside COM+ Base Services*, Mircosoft Press, 1999.

[Corra96] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Loewe, "Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach", Technical Report: TR-96-17, University of Pisa, 1996.

[Cuesta01] Carlos E. Cuesta, Pablo de la Fuenta and Manuel Barrio Solrazano, "Dynamic Coordination Architecture through the use of Reflection", *Coordination Models, Languages and Applications Special Track of ACM SAC*, 2001.

[Czarn00] Krzysztof Czarnecki and Ulrich W. Eisenecker, *Generative Programming*, Ad. Wesley 2000.

[Darwin95] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, "Specifying Distributed Software Architectures", In *Proceedings of 5th European Software Engineering Conference*, Sept. 1995.

[DSC00] Jim Dowling, Tilman Schaefer, Vinny Cahill, "Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation", In *Proceedings of Software Engineering and Reflection 2000*, LNCS 1826.

[Dow00] Jim Dowling and Vinny Cahill, "Building a Dynamically Reconfigurable minimumCORBA Platform with Components, Connectors and Language-Level Support", In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, New York, USA, April 2000.

[KLL97] Gregor Kiczales et Al., "Open Implementation Guidelines", *19th International Conference on Software Engineering (ICSE)*, ACM Press, May 1997.

[KM98] Jeff Kramer and Jeff Magee, "Analysing Dynamic Change in Distributed Software Architectures", *IEEE Proceedings – Software*, 145(5):146-154, October 1998.

[Kon01] Fabio Kon, Tomonori Yamane, Christopher K. Hess, Roy H. Campbell and M. Dennis Mickunas, "Dynamic Resource Management and Automatic Configuration of Distributed Component Systems", *USENIX COOTS'2001*.

[LeMet98] D. Le Metayer, "Describing software architecture styles using graph grammars", *IEEE Transactions on Software Engineering*, 24(7):521–553, July 1998.

[MG99] Kaveh Moazami-Goudarzi, *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD Thesis, Imperial College London, March 1999.

[MOT00] Nenad Medvidovic, Peyman Oreizy, Richard Taylor, Rohit Khare, and Michael Guntersdorfer, "An Architecture-Centered Approach to Software Environment Integration", Tech Report UCI-ICS-00-11, Dept. of Info. and Computer Science, University of California, Irvine, March 2000.

[MT00] Nenad Medvidovic and Richard N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, January 2000.

[OGT99] Peyman Oreizy et Al., "An Architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, May/June 1999.

[QuO00] Pal PP, Loyall JP, Schantz RE, Zinky JA, Shapiro R, Megquier J., "Using QDL to Specify QoS Aware Distributed Application Configuration", In *Proceedings of ISORC 2000*, March 2000.

[Rapide95] David C. Luckham and James Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734. Sep. 1995.

[SG96] M. Shaw and D. Garlan, *Software Architecture: Perspecitves on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[Silva00] Francisco Jose da Silva, M. Endler, F. Kon, Roy Campbell and Dennis Mickunas, "Modeling Dynamic Adaptation of Distributed Systems", UIUCDCS-R-2000-2196, December 2000.

[Werm00] Michel Wermelinger, *Specification of Software Architecture Reconfiguration*, PhD Thesis Universidade Nove de Lisboa, 2000.