Dynamic Software Evolution and The K-Component Model

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. There are known classes of software systems that can benefit from dynamic software evolution, including 24x7 systems that require on-line upgrades and adaptive systems that need to adapt to frequent changes in their execution environment. This paper investigates the use of dynamic software architectures and architectural reflection in building adaptive systems. We introduce the K-Component model and its architecture meta-model for building a dynamic software architecture. We address the issues of the integrity and safety of dynamic software evolution by modelling dynamic reconfiguration as graph transformations on a software architecture, and cleanly separate adaptation-specific code from functional code by encapsulating it in reflective programs called adaptation contracts. The paper also introduces the prototype implementation of our K-Component model.

1 Introduction

Computer systems that support dynamic software evolution have the ability to change their implementation at runtime allowing them to extend, customise or upgrade the services that they provide without the need for system recompilation or reboot. Designers have traditionally sought alternatives to runtime change, usually because it is avoidable. Several techniques have been devised to circumvent the need for it, including regularly scheduled downtimes, redundancy, and manual overrides. There are, however, certain classes of systems that benefit from dynamic adaptability. These include 24x7 systems, such as telecommunication switches where shutting down and rebuilding the system for upgrades may result in unacceptable delays and increased cost, and adaptive systems that adapt their provided functionality in response to the frequent changes in their usage context [Pui98]. Mobile systems, in particular, benefit from dynamic adaptability. Dynamic software evolution allows a mobile system to adapt its provided functionality in response to the often frequent changes in the device's context. There has already been much research into building middleware that supports dynamic software evolution [Blair01, Kon01, DC00].

Dynamic software architectures can be used to build dynamically evolvable software systems by supporting the self-management and reconfiguration of the system's architecture at run-time [Allen98]. Current approaches to specifying dynamic software architectures use an Architecture Description Language (ADL) [SG96] in conjunction with an Architecture Modification Language [Darwin95, Rapide95, Werm00] or a Co-ordination Language [Cuesta01]. Our approach to building a dynamic software architecture is to use *architectural reflection* [Caz00]. A system that supports architectural reflection reifies its software architecture, e.g. its configuration graph of components and connectors, as an *architecture meta-model* [DC01] 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. We also provide a separate *adaptation contract description language* for writing reflective programs called *adaptation contracts* that allow programmers to specify how and when to reconfigure the software architecture at runtime. The reconfiguration operations over the architecture are implemented as graph transformations, guaranteeing the safety and integrity of the architecture both during and after reconfiguration. In our current implementation, the reconfiguration operations allow for the replacement of components and connector strategies, but maintain a static configuration graph of the software architecture.

2 Architecture Meta-Model and 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 at runtime. In this context, behavioural reflection supports dynamic software evolution by providing the ability to rewrite a software architecture's configuration graph of components and connectors at runtime. Structural reflection [Blair01] for a software architecture is concerned with introspecting the architecture's configuration graph and constituent components, connectors and interfaces.

2.1 Architecture Meta-Model's Configuration Graph

We reify a software architecture configuration as a typed, connected graph, see Fig. 1, where the vertices are interfaces, labelled with components implementations, 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 identifiers *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* [KM98, Werm00] (see Fig. 3), is responsible for the storage and management of the software architecture's configuration graph.

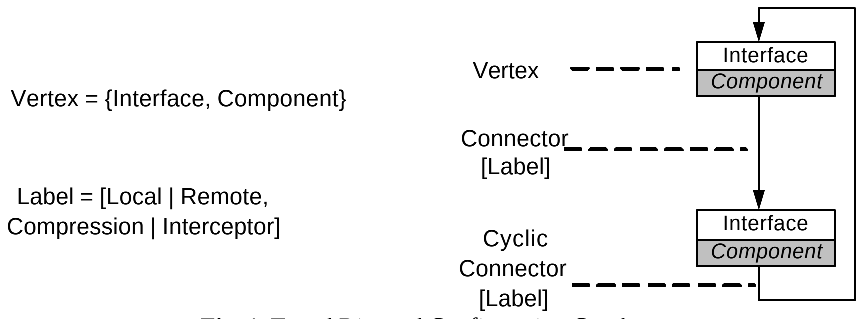

Fig. 1. Typed Directed Configuration Graph

2.2 Dynamic Reconfiguration as Configuration Graph Transformation

As mentioned previously, we model 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, see Fig. 2. Rules define how and when a graph is transformed. The interfaces and connectors that represent the vertices and edges in our graph describe the part of the system that is preserved during a graph transformation. The components and connector properties that represent the labels of the vertices and edges in our graph respectively describe the part of the graph that is rewritten during a graph transformation. Therefore, our model of dynamic reconfiguration is constrained to replacing the components in a system's configuration graph and changing the connector strategies. 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.

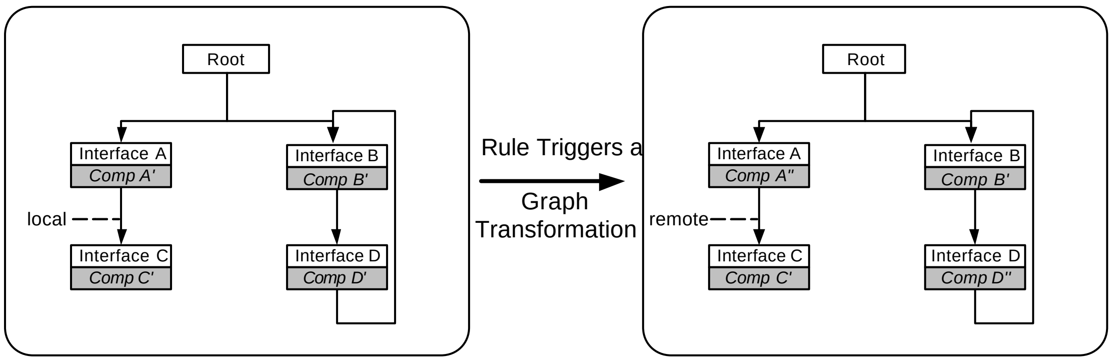

Fig. 2. Dynamic Reconfiguration of a Software Architecture as a Graph Transformation

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 rather than locking the whole graph 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]. Our reconfiguration protocol 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 [Werm00]. Component state can only be changed by computation, not by reconfiguration operations.

Our reconfiguration protocol is the following:

1. Reconfiguration operation invoked to transform the configuration graph.
2. Identify the new target configuration and sends a "freeze" message to only those components and connectors in the original configuration that will be updated in the target configuration.
3. Perform the transformation of the configuration graph, i.e. unlink, create, transfer state, remove and link the new components and/or change connector strategy.
4. Resume processing in "frozen" connectors.

We assume that a reconfiguration operation finishes in finite time and its initiator, the component manager, knows when it ends. One of the other advantages of reconfiguration protocols 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. A configuration manager is responsible for the implementation of the reconfiguration operations and the correct operation of the reconfiguration protocol.

2.3 Adaptation Contracts

To obtain a clean separation of concerns between the adaptation-specific code and the functional code, a separate language is used to specify the adaptation logic as adaptation contracts. An adaptation contract contains a series of conditional rules for the transformation of the software architecture's configuration graph. Adaptation contracts are used to specify a system's architectural constraints [Blair01], describing how and when to safely reconfigure the software architecture. 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 adaptation contracts to poll architectural constraint information from meta-level configurations and base-level components and connectors. Adaptation events also have the advantage of decoupling the meta-level adaptation contract from the base-level components and connectors. In effect, they provide a run-time separation of concerns between the adaptation code and the functional code. This allows for adaptation contracts to be dynamically loaded and unloaded, since the base-level code cannot have any direct dependencies on them.

Adaptation contracts are specified in the Adaptation Contract Description Language (ACDL). They consist of a series of conditional statements, testing for the occurrence of adaptation events, and associating reconfiguration operations with adaptation events. A configuration tool takes adaptation contracts specified in the ACDL, the software architecture specification and produces an implementation in a concrete language, i.e. C++ in our prototype. It produces an executable implementation of the software architecture, and its architecture meta-model, by creating specialised and concrete implementations of the abstract and templated classes in the K-Component framework and binding them to the software architecture. Adaptation contracts are represented at runtime by meta-level objects and are deployed in and managed by a configuration manager, see Fig. 3.

3 The K-Component Model

The K-Component architecture meta-model provides a configuration manager that stores the architecture meta-model and implements the reconfiguration (graph rewrite) operations over the architecture. The configuration manager is also a run-time container for the deployment, scheduling and execution of adaptation contracts and can optionally provide a procedural interface for the loading/unloading of adaptation contracts at runtime. It is implemented as an active object.

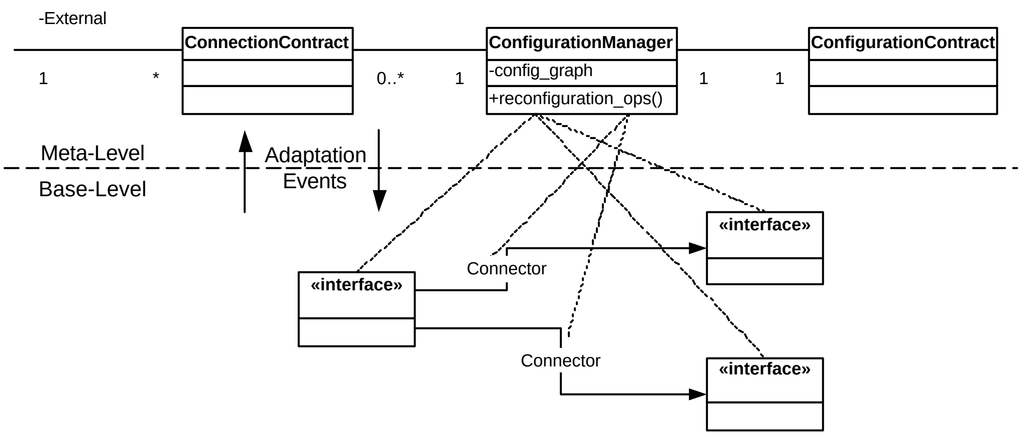

Fig. 3. Configuration Manager as a Meta-Level Adaptation Managersx

3.1 Components and Connectors

Interface Definition Language-3 (IDL-3) [CCM99] is used to define components and interfaces in the software architecture because it provides support for explicit dependency management through `provides` and `uses` interfaces. These explicit dependencies are used to help generate the configuration graph of components and connectors. IDL-3 `emits` and `consumes` events [CCM99] are used to specify adaptation events. Connectors are implemented as typed objects relating `provides` and `uses` interfaces, i.e. ports, on components. Connectors are automatically generated from IDL-3 component definitions by specialising and templating abstract connectors in the K-Component framework. 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.

3.2 Specifying the Software Architecture and Generating the Architecture Meta-Model

In our prototype implementation of K-Components, C++ is used instead of an ADL to specify the software architecture. This has the benefit of allowing the programmer to specify an application's architecture without having to learn a new language. Several abstractions and programming idioms, however, are required to represent concepts commonly found in an ADL specification, such as interfaces, connectors and binding operations. In our C++ prototype implementation for example, (component) services can only be accessed via connectors and interfaces. The following code shows how to bind a connector to an interface:

```
Connector<Interface>* connector = Factory<Interface>::bind();
```

A configuration tool is used to automatically generate the architecture meta-model by building a dependency graph from both the component definitions, in IDL-3, and the connectors, defined in the system implementation language, i.e. C++. A configuration tool 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 component instances by editing the interface labels in the XML configuration descriptor to point to actual components. Once component instances have been specified for all the interfaces in the configuration graph and adaptation contracts have been attached to the architecture, the software architecture can be instantiated by the configuration tool.

4 Future Applications – The 3-in-1 Phone

The 3-in-1 phone is an adaptable application that is based on a usage scenario for the Bluetooth intercom and cordless telephony profiles [Blu99, Blu99a]. The 3-in-1 phone dynamically adapts its software to function as three different types of phone. Within range of a Bluetooth access point (BAP), the phone functions as a cordless phone that incurs a fixed line charge. When the user leaves the range of the BAP, the phone dynamically reconfigures itself to function as a cellular phone incurring cellular charges. Finally, when the phone comes within range of another Bluetooth phone it functions as a "walkie-talkie" incurring no telephony charges.

The 3-in-1 phone can be modelled as a K-Component software architecture containing three components – the handset, the voice/multimedia component that delivers the appropriate quality of service (depending on the bit-rate of the network connection) and the network component (Bluetooth or GPRS), see Fig. 4. An adaptation contract statement specifies when to reconfigure the architecture:

```
  if (BAP_Not_Available) change_configuration(“GPRS Config”);
```

The dynamic reconfiguration of the 3-in-1 phone architecture, see Fig. 5, is triggered by an adaptation event `BAP_Not_Available`. Reconfiguration involves replacing the Bluetooth N/W component and its voice/multimedia playback component with GPRS equivalent components. The reconfiguration protocol handles the state transfer and component loading/unloading, linking and unlinking.

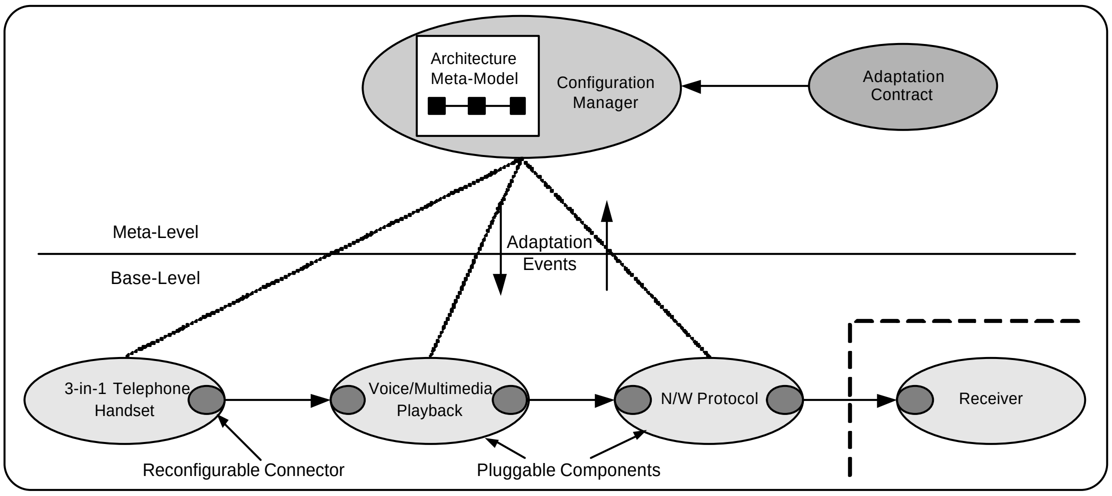

Fig. 4. 3-in-1 Phone Application as a K-Component Application

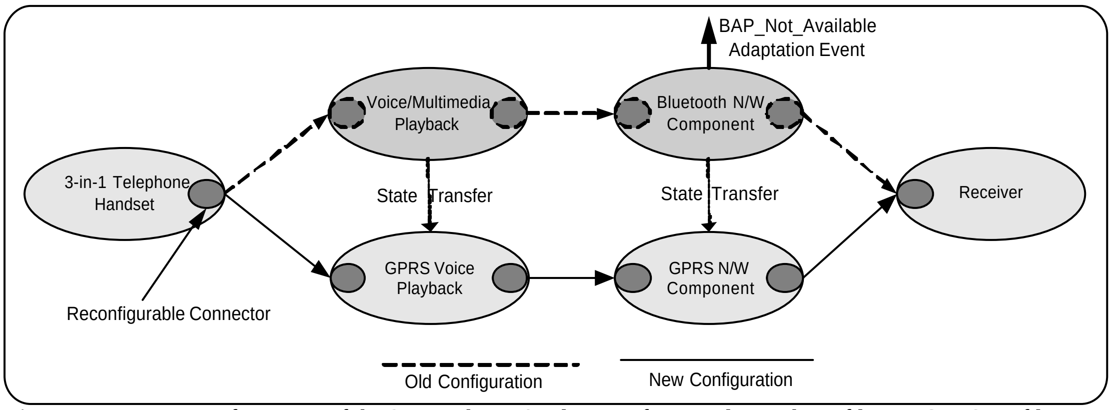

Fig. 5. Dynamic Reconfiguration of the 3-in-1 Phone Application from a Bluetooth Profile to a GPRS Profile

Conclusions

Software architecture concepts, such as inter-object dependencies, have always been present in object-oriented software, but their importance has increased with the advent of dynamically evolvable software. In this paper, we have presented the K-Component model as a framework for explicitly reifying the software architecture in object-based component systems as an architecture meta-model. We provide support for safely reconfiguring the architecture using reflective programs, called adaptation contracts, that perform conditional graph transformations on the architecture. We also provide the adaptation contract description language for a clean separation of adaptation-specific code from functional code. The K-Component model can be used to build adaptive applications.

Bibliography

[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.

[Blu99] Bluetooth Consortium, K4: Intercom Profile, Bluetooth Specification Version 1.0, 1999.

[Blu99a] Bluetooth Consortium, K3: Cordless Telephony Profile, Bluetooth Specification Version 1.0, 1999.

[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.

[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.

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

[DC01] Jim Dowling and Vinny Cahill, “The K-Component Architecture Meta-Model for Self-Adaptive Software”, *Reflection 2001 The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan September 25-28, 2001.

[DC00] 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.

[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*.

[OGT99] Peyman Oreizy , Michael M. Gorlick, Richard N. Taylor, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf, “An Architecture-Based Approach to Self-Adaptive Software”, *IEEE Intelligent Systems*, May/June 1999.

[Pui98] Salber, D., Abowd, G. "*The Design and Use of a Generic Context Server*," Technical Report GIT-GVU-98-32, Georgia Institute of Technology, 1998.

[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.

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