Managing Dynamic Reconfiguration in Component-Based Systems

Thais Batista[1,2,*], Ackbar Joolia[2], and Geoff Coulson[2]

[1] Computer Science Department,
Federal University of Rio Grande do Norte (UFRN),
59072-970, Natal – RN, Brazil
thais@ufrnet.br
[2] Computing Department, InfoLab21, Lancaster University,
LA1 4WA, Lancaster, UK
{t.batista, a.joolia, g.coulson}@lancaster.ac.uk

Abstract. We propose a meta-framework called 'Plastik' which i) supports the specification and creation of runtime component-framework-based software systems and ii) facilitates and manages the runtime reconfiguration of such systems while ensuring integrity across changes. The meta-framework is fundamentally an integration of an architecture description language (an extension of ACME/Armani) and a reflective component runtime (OpenCOM). Plastik-generated component frameworks can be dynamically reconfigured either through programmed changes (which are foreseen at design time and specified at the ADL level); or through ad-hoc changes (which are unforeseen at design time but which are nevertheless constrained by invariants specified at the ADL level). We provide in the paper a case study that illustrates the operation and benefits of Plastik.

1 Introduction

Software architecture modeling using Architecture Description Languages (ADLs) is becoming increasingly popular in the *early phases* of system development [1, 2, 3]. Such languages facilitate the construction of high-level models in which systems are described as compositions of components. They play an important role in developing high quality software by supporting reasoning about structural properties early in the development process. This can make it easier to produce more extensible structures, locate design flaws, and better maintain consistency.

At the same time there has been a parallel development of *runtime* component models which are targeted at the actual construction and deployment of systems [4,5,6,7]. These component models are becoming quite sophisticated in their capabilities for runtime reconfiguration. For example, they use reflective or runtime aspect-oriented programming techniques to allow software to inspect, adapt and extend itself while it is running. This is particularly useful in inherently adaptive software environments such as mobile computing and adaptive real-time systems [8].

[*] Thais Batista is supported by the Brazilian Research Council (CAPES) project BEX0680/04-4.

R. Morrison and F. Oquendo (Eds.): EWSA 2005, LNCS 3527, pp. 1 – 17, 2005.
© Springer-Verlag Berlin Heidelberg 2005

It is clear that an integration of the two above-mentioned strands of development holds significant potential. Some early work has attempted to do this (see related work discussion in section 5) but this has typically suffered from two main limitations: *i*) it has not taken a sufficiently comprehensive approach to formally specifying and constraining runtime reconfiguration at the ADL level, and *ii*) it has not leveraged the most recent developments in reconfigurable runtime component technologies. The 'Plastik' meta-framework described in this paper is an ADL/ component runtime integration that attempts to address such limitations.

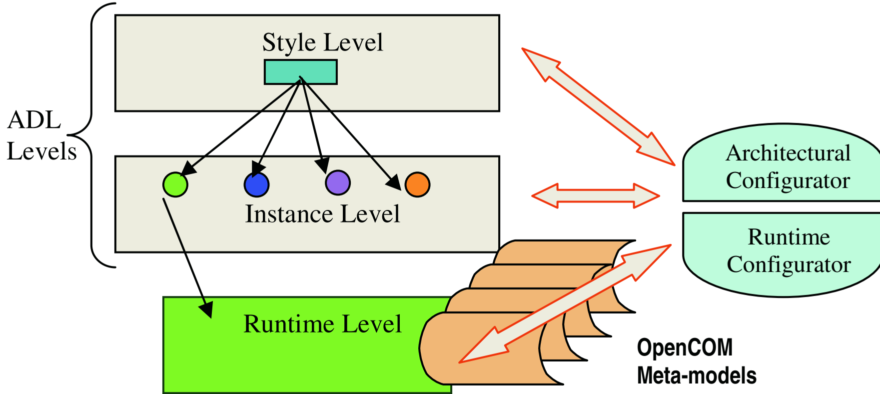

Fig. 1. Plastik's system architecture

The Plastik architecture, illustrated in figure 1[1], supports formally-specified runtime reconfiguration of systems through an integration of an ADL and a reflective, component model runtime. The ADL level is based on ACME/ Armani [9,10] which we have enhanced with new constructs for dynamic reconfiguration; and the runtime level is based on our OpenCOM component model [7] and its association notions of component frameworks and reflective meta-models [12].

Plastik supports both programmed and ad-hoc reconfiguration:

- *Programmed reconfiguration* pertains to changes that can be foreseen at system design time. In Plastik, this is supported at the ADL level in terms of 'predicate-action' specifications. For example, consider a PDA-based video application that needs to run over both fixed and wireless networks. In such an environment, one could specify a programmed reconfiguration that switches from a MPEG decoder to an H.263 decoder ('action') when the PDA detects a drop in the quality of network connectivity ('predicate') [12]. In this example, the predicate could be expressed in our extended ADL as a function of a dynamic property of an underlying protocol component, and the associated action would take the form of (extended) ADL statements that replace the old component with the new one.

[1] Section 3 expands on the various entities depicted in figure 1.

- *Ad-hoc reconfiguration*, on the other hand, is intended for changes that are not and cannot be foreseen at system design time. The approach here is to build general invariants into the specification of the system and to accept any change as long as the invariants are not violated. As an example, the above mobile computing scenario might be enhanced by the insertion of a jitter-smoothing buffer which, despite not having been considered at design time, could nevertheless be usefully inserted at runtime.

In addition, Plastik allows both programmed and ad-hoc reconfiguration to be initiated from multiple architectural levels (see section 3) which enables considerable flexibility.

The remainder of this paper is structured as follows. Section 2 provides background on ACME/ Armani and on the OpenCOM component runtime; it also considers the relationship between the two technologies. Section 3 then details our approach to programmed and ad-hoc reconfiguration, and section 4 presents a case study which exemplifies the approach. Finally, section 5 discusses related work, and section 6 offers our conclusions.

2 Background on ACME/Armani and OpenCOM

2.1 The ACME/Armani ADL

The Plastik meta-framework's ADL level provides the basis for specifying systems and enabling and constraining their reconfiguration. We have selected ACME [9] as our ADL because:

- Unlike many ADLs it offers sufficient generality to straightforwardly describe a variety of system structures. Most ADLs are domain-specific so they do not provide generic structures to cope with a wide range of systems.
- It comes with tools that provide a good basis for designing and manipulating architectural descriptions and generating code.

The basic elements of ACME are as follows: *Components* are potentially composite computational encapsulations that support multiple interfaces known as *ports*. Ports are bound to ports on other components using first-class intermediaries called *connectors* which support so-called *roles* that attach directly to ports. *Attachments* then define a set of port/role associations. *Representations* are alternative decompositions of a given component; they reify the notion that a component may have multiple alternative implementations. The ACME type system provides an additional dimension of flexibility by allowing type extensions via the *extended with* construct. *Properties* are <*name*, *type*, *value*> triples that can be attached to any of the above ACME elements as annotations (apart from attachments). Finally, *architectural styles* define sets of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed in a reusable architectural domain (see example below).

In addition, we adopt the *Armani* [10] extensions to ACME. Armani is a FOPL-based sub-language that is used to express architectural constraints over ACME archi-

tectures. For example, it can be used to express constraints on system composition, behavior, and properties. Constraints are defined in terms of so-called *invariants* which in turn are composed of standard logical connectives and Armani predicates (both built-in and user-defined) which are referred to as *functions*. Although Armani appears to introduce an element of dynamicity, it is important to emphasise that ACME/Armani does *not* currently support dynamic runtime reconfiguration of systems (see also section 5).

The ACME fragment below illustrates the main ACME/Armani concepts. The style definition includes two port types and two roles types. The *OSIComp* component type then defines the central player in a layered communications system environment. This definition includes a connector type which is used to connect protocol layers and an Armani invariant that states that a system must comprise a four-level stack.

```
Style PlastikMF {
          Port Type ProvidedPort, RequiredPort;
          Role Type ProvidedRole, RequiredRole;
          ….
};

Component Type OSIComp: PlastikMF {
    ProvidedPort Type upTo, downTo;
    RequiredPort Type downFrom, upFrom;

    Property Type layer =
       enum {application, transport, network, link};
};

Connector Type conn2Layers: PlastikMF {
    ProvidedRole Type source;
    RequiredRole Type sink;
};

Invariant
    Forall c:OSIComp in sys.Components
          cardinality(c.layer = application) = 1 and
          cardinality(c.layer = transport) = 1 and
          cardinality(c.layer = network) = 1 and
          cardinality(c.layer = link) = 1 and

Property Type applicationProtocol;
Property Type transportProtocol;
Property Type networkProtocol;
Property Type linkProtocol;
```

Fig. 2. An example definition in ACME

2.2 The OpenCOM Reflective Component Model

The OpenCOM component runtime has been extensively used over the past few years to build reconfigurable systems software elements such as middleware and programmable networking environments [11]. A high-level view of its programming model is given in figure 3. Components (the filled rectangles) are encapsulated units of functionality and deployment that interact with their environment (i.e. other components) exclusively through interfaces (the small circles) and receptacles (the small cups). A component may support multiple interfaces and receptacles and may be internally composite (i.e. composed of other components). Components are deployed at runtime into environments called capsules (the outer dotted box) which support a runtime 'capsule API' containing operations to load/ unload components (and also to bind/ unbind interfaces and receptacles; see below). The loading of components into a capsule can be requested by any component inside or outside the capsule (this is referred to as third-party deployment). Interfaces are units of service provision offered by components; they are expressed in terms of sets of operation signatures and associated datatypes. For programming language independence, OMG IDL is used as an interface definition language. Receptacles are 'anti-interfaces' used to make explicit the dependencies of components on other components: whereas an interface represents an element of service provision, a receptacle represents a unit of service requirement. Receptacles are key to supporting a third-party style of composition (to complement the third-party deployment referred to above): when third-party deploying a component into a capsule, one knows by looking at the component's receptacles precisely which other component types must be present to satisfy its dependencies. Finally, bindings, which are created via the capsule API, are associations between a single interface and a single receptacle. As with loading, the creation of bindings is inherently third-party in nature; it can be performed by any party inside or outside the capsule.

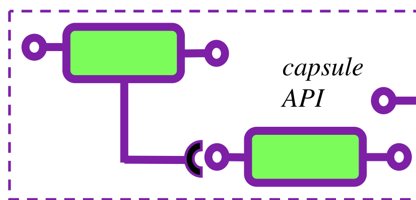

Fig. 3. The OpenCOM component model

In implementation, the OpenCOM programming model is supported by a small runtime of around 17KB in size. Components are written in C++ by default. As well as supporting the programming model concepts described above, the OpenCOM runtime also supports a set of so-called *reflective meta-models* [12] which facilitate reconfiguration of systems by permitting different system aspects to be inspected, adapted and extended at runtime. In particular, OpenCOM employs the following meta-models:

- an *architecture meta-model* which exposes the compositional topology of a system of deployed components in terms of a causally-connected graph structure;
- an *interception meta-model* which allows one to interpose interceptors at bindings between component interfaces; and
- an *interface meta-model* which allows one to discover information about interfaces at runtime and to invoke interface types that are dynamically discovered at runtime.

The final key aspect of OpenCOM is that it supports building systems in terms of the medium granularity (i.e. between components and whole systems) notion of *component frameworks* (hereafter, CFs) [12]. CFs are tightly-coupled clusters of components that cooperate to address some focused domain of functionality, and which accept 'plug-in' components that tailor or extend functionality in that domain. The idea is that one constructs systems by composing and configuring appropriate CFs. For example, one might develop a middleware system by composing CFs that address independent functionality domains such as protocol stacking, thread scheduling and request-handling [12]. Importantly, CFs incorporate policies and constraints that determine how and to what extent the CF can be runtime reconfigured. Typically, per-CF constraints are also imposed on the use of the reflective meta-models. Essentially, reflection provides maximal openness and flexibility, whereas CFs channel and constrain this expressive 'power' into useful and safe forms.

2.3 Mapping from ACME/Armani to OpenCOM

As can readily be observed, there is a close correspondence between concepts in ACME/Armani and in OpenCOM. This correspondence is summed up in table 1.

Table 1. ACME/Armani to OpenCOM correspondences

ACME/Armani	OpenCOM
component	*(composite) component*
connector	*(composite) component*
port	*interface/ receptacle*
role	*interface/ receptacle*
attachment	*binding*
representation	*(composite) component*
property	*interface operation*
style	*CF*
invariant	*CF constraints*

The style-to-CF correspondence is central. As domain-specific units of re-usable and dynamically reconfigurable functionality, OpenCOM CFs are the natural target abstraction for ADL-specified styles whose specification incorporates programmed reconfiguration and constraints on ad-hoc reconfiguration. This observation forms the

basis for Plastik's approach to reconfiguration as detailed in the next section. The fact that OpenCOM supports third-party deployment and binding is also crucial in enabling the runtime to be manipulable from the ADL level.

3 Approach to Reconfiguration

3.1 Architecture

Before discussing our approach to programmed and ad-hoc reconfiguration, we briefly expand on the architecture diagram presented in figure 1.

Note first that figure 1 has *two* ADL sub-levels: a style level and an instance level. The style level is used to define generic patterns—an example could be a 'protocol stacking' style which defined a basic set of elements and constraints for describing linear compositions of 'protocol' components. The instance level then particularises a style for a specific context while honouring any constraints imposed by the style. For example, one could define an 'TCP/IP stack' CF that imposed the additional constraints that the maximum number of levels was 4, that a stack can only be reconfigured when a connection is dormant, and that a "TCP" component must always be placed above an "IP" component.

Figure 1 also illustrates Plastik's *system configurator* which is divided into two levels: an *architectural configurator* responsible for accepting and validating reconfiguration requests at the ADL levels, and a *runtime* configurator responsible for managing the OpenCOM/ runtime level. There is one instance of the architectural configurator in the whole Plastik system, but there is one instance of the runtime configurator for each deployed CF. Both parts of the configurator are implemented in an interpreted scripting language called Lua [13]. The link between the ADL and the runtime levels is realised as an ACME/ Armani compiler (we use AcmeLIB [22] as the basis of this). The output of the compiler is a Lua program that instantiates OpenCOM elements that correspond to the ADL-level specifications. The compiler also generates finite state machines that implement Armani invariants as discussed below. These are located in the runtime configurator of each CF. More detail is given below.

3.2 Programmed Reconfiguration

3.2.1 Limitations of ACME/ Armani

As indicated in the introduction, we address programmed reconfiguration by providing appropriate extensions to ACME/Armani. Before introducing these extensions, we will briefly motivate them by analysing the limitations of 'standard' ACME/ Armani with respect to dynamic programmed reconfiguration.

Programmed reconfiguration *could potentially* be expressed using the following existing ACME/Armani concepts as a basis:

- The Armani 'invariants' are potentially useful in ensuring that a system preserves the constraints imposed by the software architecture despite the dynamic insertion or removal of ACME elements.
- The 'extend with' construct enables type extension, and could conceivably be applied to extend types at runtime.

- The ‘representation’ construct could be used as a basis of switching from one representation of a component to another at runtime.
- The ‘properties’ construct could also be used to describe how components may be changed at runtime.

Nevertheless, these features are insufficient as a basis for runtime reconfiguration. First, the ‘extend with’ and ‘representation’ constructs do not address the most general reconfigurations that might be required—e.g. those involving removal of components or other elements. Second, ‘properties’ on their own are severely limited by the fact that they have no inherent semantics—which means that their interpretation is intuitive and depends on a shared understanding. Furthermore, neither properties nor any of the other constructs mentioned provide any way of specifying *when* reconfiguration should take place or *what* should be changed in any particular configuration operation.

3.2.2 ACME Extensions for Programmed Reconfiguration

The first extension is a conditional construct that allows the ADL programmer to express runtime conditions under which programmed reconfigurations should take place, together with a specification of what should change. The syntax of the construct is as follows:

```
On (<predicate>) do <actions>
```

The ‘predicate’ is expressed using the standard Armani predicate syntax, and refers to properties attached to ACME components. Composite predicates involving multiple properties are supported. As will be explained later, it is these properties that ‘ground’ the predicate in the OpenCOM runtime system. The ‘actions’ are arbitrary ACME statements[2] that are instantiated when the predicate becomes true. These statements could, for example, declare additional components and connect them into the existing architecture by declaring additional attachments. Where more than one action is specified, it is assumed that the set of actions will be instantiated in sequence and atomically.

The second extension is a pair of constructs that specify the destruction of existing ACME elements:

```
detach <element>
remove <element>
```

Detach is used to remove an attachment between a port and a role; and *remove* is used to destroy an existing component, connector or representation. Removal of elements is only possible when they are no longer involved in an attachment. The idea is that *remove* and *detach* can be used as *On-do* actions to enable architectures to be dismantled as well as constructed. Given this capability, fully general runtime changes are possible, ranging from simple replacement of an element to a wide-ranging reconfiguration that can modify the whole architecture. The use of *remove* and *detach* in conjunction with *On-do* is illustrated in figure 4:

---

[2] In this and the following extensions, we build on existing ACME constructs but apply them (such as here) in novel syntactical contexts. The semantics are, however, maintained.

```
On (net_bandwidth = low) do {
   detach MPEG-dec.req to conn-dec.p;
   remove MPEG-dec;
   Component H263-dec : decoder = new decoder extended with {
       Property decoder-type = "H263";
   };
   Attachments
           H263-dec.r to conn-dec.p;
}
```

Fig. 4. Example of use of the *On-do* statement

When the given predicate becomes true (i.e. when the *net_bandwidth* property transits to the value *low*), the following reconfiguration sequence takes place: component *MPEG-dec* is detached from connector *conn-dec* and removed; and then a new H.263 component is instantiated and attached to the same connector.

The third extension that we propose is intended to express runtime dependencies between architectural elements. Managing dependencies among first-class entities is especially important to dynamic reconfiguration to avoid architectural mismatches when a new element is inserted in a system. The syntax of this extension is as follows:

```
dependencies <statements>
```

The *dependencies* statement allows expression of the fact that dynamic instantiation/ destruction components is dependent on the creation/ destruction of other components. Here is an illustration of the use of *dependencies*:

```
Component transport: OSIComp = {
    ...
    dependencies {
          extended with {RequiredPort bufport};
          Component bm: bufferManager;
          Invariant
          forall p:ProvidedPort in bm.Ports
                p.rate > 1000
          }

          Connector transtobuf {
          ProvidedRole pr;
          RequiredRole rr;
          }
          Attachments {
          transport.bufport to transtobuf.rr;
                bm.pp to transtobuf.pr;
          }
    }
}
```

Fig. 5. Example of use of the *dependencies* statement

This specifies that the *transport* component depends on a buffer manager; therefore an instance of the latter is instantiated and attached whenever an instance of the former is created. (The example also includes an invariant that requires that the buffer manager must be able to accept data at a certain rate.)

The fourth and final extension allows attachments to be specified for a *type* as well as for an instance (only instances are supported by standard ACME). The precise instance to be used is selected at runtime according to a policy specified by the associated connector[3]. The syntax of the construct is as follows:

```
<connector> to dynamic <componentport>
```

An example of the use of the dynamic statement is shown in figure 6. This assumes that *ConnX* contains a policy that determines which instance of the *Network* component type will be attached.

```
Attachments {ConnX.r to dynamic Network.p}
```

Fig. 6. Example of use of the *dynamic* statement

Note that there is an analogue to this sort of dynamic component instantiation in Darwin [14]. However, in Darwin it is not possible to declare an attachment to a specified 'provided' port of a dynamic component. We consider that this makes the architectural description unclear and can lead to unexpected bindings at runtime.

3.2.3 Supporting Programmed Reconfiguration at Runtime

This largely amounts to providing runtime support for the above-described ACME extensions for programmed reconfiguration. First, the predicate element of each *On-do* statement is compiled into a runtime finite state machine (FSM) representation. All the FSMs for each Plastik CF are contained in the associated per-CF runtime configurator. As mentioned, the ADL-level predicates are 'grounded' into the OpenCOM runtime through their embedded property elements. In particular, it is required that each ADL-level property is supported by corresponding 'property operations' in a distinguished interface of the OpenCOM component that underpins the ADL-level component to which the property is attached. There are simple lexical conventions that tie ADL-level property names to runtime level property operations, and the property operation are discovered and bound to by the configurator at runtime using OpenCOM's interface meta-model. Given this machinery, the FSMs are evaluated every time a runtime property operation reports (via a callback) a change in the value of the runtime property. This evaluation may then trigger an execution of the *On-do* statement's 'actions' clause.

Execution of the actions clause is carried out transactionally in case the whole sequence cannot be completed (e.g. if an attempt is made to remove an element that is still attached to some other element). It is also important to confirm that the proposed reconfiguration will not violate any general Armani-specified constraints elsewhere in the CF (whether at the style or the instance levels). These general constraints are discussed further in section 3.3.

[3] We are also considering alternative means of specifying these policies.

Implementation of the *detach* and *remove* actions make use of OpenCOM's architecture meta-model to ensure that the required preconditions of these actions (see section 3.2.2) are satisfied. The load/unload and bind/unbind APIs of OpenCOM's capsule API are then used to effect each actions. The *dependencies* statement causes the runtime to dynamically load (and bind) any dependent components whenever it instantiates an OpenCOM component whose ADL-level analogue specifies such a dependency. Finally, the implementation of the *dynamic* construct also builds directly on OpenCOMs load/ bind APIs. It involves the prior evaluation of an associated policy to select the appropriate instance—this is performed by Lua code generated from the policy statement.

3.3 Ad-Hoc Reconfiguration

By definition, ad-hoc reconfiguration is not specified at the ADL level. Rather, our approach is to *constrain* at the ADL-level the allowable range of permissible ad-hoc reconfigurations. For this we again rely on Armani invariants and similarly ground the invariants using property values that refer to the runtime level.

In Plastik, ad-hoc reconfiguration can be initiated either at the ADL level or at the runtime level. ADL level ad-hoc reconfiguration involves submitting an *architecture modification script* to the architectural configurator. This script is written in our extended ACME and specifies a set of proposed runtime changes to a target ADL specification. The script may not include invariants at the top level. The changes are applied to the target specification which is then recompiled to produce a Lua *diff script* that is (transactionally) executed to reconfigure the runtime CF. As in the case of programmed reconfiguration, the runtime system confirms before making any changes that running the diff script will not violate any architectural constraints specified in the target ADL specification. Both style and instance level invariants are taken into account. Notice that because the architecture modification script is written in extended ACME, it is possible to dynamically add new programmed reconfiguration clauses to a running CF.

Reconfiguration requests at the runtime level take the form of operations directly applied the OpenCOM reflective meta-models. This is the 'traditional' means of exploiting OpenCOMs reconfiguration capabilities. In Plastik, however, the meta-model APIs are hidden by automatically-generated per-CF wrappers so that calls on them are first validated by an evaluation of the invariants as discussed above.

Supporting ad-hoc reconfiguration at both the ADL and runtime levels raises issues of *causality*—i.e. to what extent are changes at one level reflected in the other? Our current approach is to provide full causality in the ADL-to-runtime direction, but not in the other direction. An implication of this is that a runtime-level ad-hoc reconfiguration may cause rejection of a subsequent ad-hoc reconfiguration at the ADL level due to an inconsistency having being introduced—e.g. if the ADL-level reconfiguration request refers to some component that has previously been removed by the runtime-level reconfiguration. In practice, we expect that most CFs will employ *either* ADL-level *or* runtime-level ad-hoc reconfiguration but not both. Use of the runtime level is appropriate in low-level system environments that are driven primarily by

dynamic events in other low-level CFs. Use of the ADL-level, on the other hand, is more appropriate for higher-level CFs that are primarily driven by applications or GUIs.

4 Case Study

To further illustrate the use of Plastik, we extend the running *OSIComp* example to demonstrate both programmed and ad-hoc reconfiguration of the example protocol stack illustrated in figure 7.

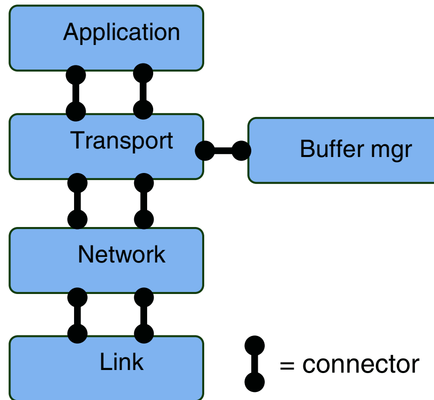

Fig. 7. Example of a reconfigurable protocol stack

4.1 Programmed Reconfiguration

To illustrate programmed reconfiguration consider changing the Application component of our system from an MPEG decoder to an H.263 decoder on the basis of a change in available bandwidth (as outlined in the introduction and specified in figure 4). The definition of the MPEG component, which derives from *decoder* which in turn derives from *OSIComp*, is as follows:

```
Component Type decoder:OSIComp = new OSIComp extended with
{};
Component MPEG-dec: decoder = new decoder extended with {
  ProvidedPort transportProtocol:downTo = {
    Property protocol:string="tcp";
    Invariant
       Forall p in self.ProvidedPorts
                    p.protocol = transportProtocol
  };
  Property layer = 'application';
  Property decoder-type="MPEG";
};
```

(Note the invariant that requires that the component can only be connected to a transport protocol.) The programmed reconfiguration is specified in the below definition of the complete system.

```
System OSIStack : PlastikMF {

Component MPEG-dec:OSIComp; // Application Level
Component Transport:OSIComp;
Component Network:OSIComp;
Component Link:OSIComp;

Connector AppToTrans: conn2Layers;
Connector TransToNet: conn2Layers;
Connector NetToPhys: conn2Layers;

Attachments{
     //connecting the Application to Transport layer
     Application.dataTo to AppToTrans.source;
     Transport.dataFrom to AppToTrans.sink;

     //connecting the Transport to Network layer
     Transport.dataTo to TransToNet.source;
     TransToNet.sink to dynamic Network.dataFrom;

     //connecting the Network to Physical layer
     dynamic Network.dataTo to NetToPhys.source;
     Link.dataFrom to OuterApplication;
};
On (Link.net_bandwidth = low) do
{
     detach MPEG-dec.downTo to AppToTrans.source;
     remove MPEG-dec;
     Component H263-dec : decoder = new decoder extended with
     {
        Property decoder-type = "H263";
      };
     Attachments
            H263-dec.downTo to AppToTrans.source;
};
};
```

The key part of this is the *On-do* statement, the predicate of which includes a *Link.net_bandwidth* property. This is a property of the link layer component and, as outlined in section 3.2.3, is realised at the runtime level as a dynamic 'property operation'. Depending on its value, this property will trigger a programmed reconfiguration that replaces the MPEG-decoder component with an H.263-decoder component.

4.2 Ad-Hoc Reconfiguration

As mentioned, ad-hoc reconfiguration can be initiated from either the ADL level or the runtime level.

As an example of ad-hoc reconfiguration at the ADL level consider changing the Transport component's BufferManager with a larger BigBufferManager. The change script to achieve this is as follows:

```
//inserting new Component BigBufferManager
Component BigBufferManager{
  ProvidedPort pp;
  RequiredPort rp;
      …
};

detach BufferManager.pp to transtobuf.pr;
remove BufferManager;

Component bbm: BigBufferManager = new BigBufferManager;
Attachments{
      BigBufferManager.pp to transtobuf.pr;
};
```

The script detaches and removes the old component, and then creates and attaches an instance of the new component.

As an example of reconfiguration at the runtime level consider inserting a logging component between the Network and the Link layers. This could be implemented using the OpenCOM meta-models [7] as follows.

```
Component_instance loggingI;
Loaded_component logging; //new component to be loaded

logging = load(comp_type_logging);
loggingI = instantiate(logging);
//use the Architecture meta-model to inspect and insert the
Logging component

if(ArchMM.connected(Network-comp, Link-comp))
{
      ArchMM.unbind(Network-comp,Link-comp);
      ArchMM.bind(NEWBINDER,Network-comp,loggingI);
      ArchMM.bind(NEWBINDER,loggingI,Link-comp);
      ArchMM.insert(loggingI,CLSID);
      ArchMM.updateLink(CLSID,Network-Comp);
}
```

This pseudo-C code uses the architecture meta-model to discover the current topology of the system and then uses the OpenCOM's capsule API to insert the logging component. Recall that the calls to the meta-model are 'wrapped' by Plastik so that it can be ensured that they do not break any architectural constraints that were specified at the ADL level.

5 Related Work

Relevant areas of related work are as follows: software architecture, frameworks that support reconfiguration, and component runtime systems.

Software Architecture. Dynamic ACME [15] is an ACME extension that models dynamic architectures. However, it is focused on constraining evolution of specifications rather than providing support for runtime reconfiguration.

ArchWare [16] shares some similarities with Plastik as it implements dynamic change via reflection and reification, and is driven by an ADL with formal support. ArchWare uses *hyper-code*, an active executing graph with a programmable interface, as a representation, for purposes of reflection, of the executing system.. In contrast, Plastik adopts a efficient component runtime as its execution element and focuses on the mapping from an (extended) ADL to this runtime.

Mae (Managing Architectural Evolution) [17] is an architectural evolution environment that uses xADL to specify architectures. Its basis for reconfiguration is a versioning mechanism combined with a check-out/check-in approach. A key difference between this work and ours is that Mae supports only programmed reconfiguration (it achieves this by selecting architectural configurations from a 'version space'). It also lacks a formal approach with which to impose constraints to ensure consistency upon reconfiguration..

Frameworks that Support Reconfiguration. [18] focuses on evolution guided by the idea that architectures must react to events and perform architectural changes autonomously. 'Agents' receive external events, monitor the global architecture, and capture and manage changes in the architecture. Each agent maintains a knowledge base with information about the architecture and rules for programmed reconfiguration. The 'B' formalism is used to specify the architectural representation and constraints. This work has some similarities with Plastik in the sense they both use ADL and formal methods as a basis for implementing reconfiguration. Unlike Plastik, however, this work does not use reflection to implement dynamic reconfiguration and ad-hoc reconfiguration is restricted because it is based on a-priori defined rules.

FORMAware [19] is a reflective component-based framework that combines explicit architectural description and meta-information to constrain reconfiguration. To avoid inconsistency it checks architectural constraints according to style rules that restrict the types of architecture elements and possible configurations. A transaction service manages the reconfiguration. A fundamental difference between our work and FORMAware is that our proposal includes statements to improve ADL expressiveness for defining ad-hoc and programmed dynamic reconfiguration. In addition, unlike FORMAware, we adopt a formal approach to set constraints and ensure consistency upon reconfiguration.

Jadda (Java Adaptive component for Dynamic Distributed Architecture) [20] is another framework that relies on architecture specification to support dynamic reconfiguration. It uses xADL and again no formal support is provided for constraining dynamic reconfiguration. Jadda's support for ad-hoc reconfiguration is accomplished via a console that is used to submit a xADL file with the change specification. Although it handles dynamic architectural changes, Jadda is limited to ad-hoc reconfiguration with no formal support. Thus, it does not guarantee consistency.

Component Runtime Systems. Fractal is a hierarchically-structured component model [5] that provides reflective features to support dynamic architectural reconfiguration. It uses an XML-based ADL to specify the high level structure of an application. Although this work resembles our proposal in outline it does not support ad-hoc reconfiguration nor define expressive constructs at the ADL level to describe reconfiguration possibilities. In addition, the ADL has no formal support to ensure consistency. Moreover, the relationship between the architecture level and the component runtime is not clearly specified in the literature.

Finally, Koala [21] is a component model that uses an ADL based on Darwin to manage the complexity of software in electronics products. However, dynamic reconfiguration is restricted to switching between components based on statically defined conditions. Moreover, changes in component structure need administrator approval.

6 Conclusions

We have proposed a meta-framework that relies on a style-based ADL associated with a formal approach to describe the architecture and behavior of systems. It directly supports programmed reconfiguration and also provides invariants that constrain ad-hoc reconfiguration. The ADL level is supported by a flexible configurable component runtime which grounds the ADL level in a viable implementation environment. The paper focuses on extensions to ACME/Armani that express both programmed and ad-hoc reconfiguration. It also outlines the mapping from the ADL description to the OpenCOM component runtime entities and shows how ad-hoc changes can be initiated from either the ADL or the runtime level.

Currently, we are using the AcmeLIB tools to implement the compiler and runtime FSM engines discussed in section 3. At the time of writing we do not have a fully implemented system but rather have successfully trialed key aspects of the design.

Planned future work includes investigating further the issue of causality between changes made at the different architectural levels (see section 3.3), and carrying out experiments with more realistic application scenarios.

References

1. Shaw, M. and Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, (1996)
2. Allen, R. J. and Douence, R. and Garlan, D.: Specifying and Analyzing Dynamic Software Architecture In: Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98). March (1998).
3. van der Hoek, A., Heimbigner, D., Wolf, A.: Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois. Technical Report CU-CS-849-98, University of Colorado, (1998).
4. Fassino, J., Stefani, J-B., Lawall, J. and Muller, G.: THINK: A Software Framework for Component-based Operating Systems Kernels. In USENIX’02, pages 73-86, Monterey, CA, USA, (2002).

5. Bruneton, E., Coupaye, T., Stefani, J-B.: Recursive and Dynamic Software Composition with Sharing. Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain, (2002).
6. Zachariadis, S., Mascolo, C., Emmerich, W.: Satin: A Component Model for Mobile Self Organisation. Distributed Objects and Applications (DOA), pages 1303-1321, (2004).
7. Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J.: OpenCOM v2: A Component Model for Building Systems Software, Proceedings of IASTED Software Engineering and Applications (SEA'04), Cambridge, MA, ESA, Nov (2004).
8. Yau, S., Karin. F.: An Adaptive Middleware for Context-Sensitive Communications for Real-Time Applications in Ubiquitous Computing Environments. Real-Time Systems, 26(1):29-61, January (2004)
9. Garlan, D. and Monroe, R. and Wile, D.: ACME: Architectural Description of Component-based Systems. Foundations of Component-based Systems. Leavens, G. T., and Sitaraman, M. (eds). Cambridge University Press, pp. 47-68, (2000).
10. Monroe, R. T.: Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University.
11. Coulson, G., Blair, G.S., Hutchison, D., Joolia, A., Lee, K., Ueyama, J., Gomes, A.T., Ye, Y.: NETKIT: A Software Component-Based Approach to Programmable Networking, *ACM SIGCOMM Computer Communications Review (CCR)*, Vol 33, No 5, pp 55-66, October (2003).
12. Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N.: The Design of a Highly Configurable and Reconfigurable Middleware Platform, ACM Distributed Computing Journal, Vol 15, No 2, pp 109-126, April (2002).
13. Ierusalimsky, R., Figueiredo, L. H., and Celes, W.: Lua – an extensible extension language. Software: Practice and Experience, 26(6):635-652, (1996).
14. Magee, J., Dulay, N., Eisenbach, S. and Kramer, J.: Specifying Distributed Software Architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, pp. 137-153, September (1995).
15. Wile, D.: Using Dynamic Acme. In: Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December, (2001).
16. Morrison, R. et al.: Support for Evolving Software Architectures in the ArchWare ADL. In: *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA), Oslo, Norway*. 2004.
17. Roshandel, R., van der Hoek, A., Mikic-Rakic, M. and Medvidovic, N.: Mae – A System Model and Environment for Managing Architectural Evolution. ACM Transactions on Software Engineering and Methodology (TOSEM), 3 (2):240-276, (2004).
18. Ramdane-Cherif, A. and Levy, N.: An Approach for Dynamic Reconfigurable Software Architectures. Integrated Design and Process Technology, IDPT-2002, June (2002).
19. Moreira, R., Blair, G., Carrapatoso, E.: FORMAware: Framework of Reflective Components for Managing Architecture Adaptation, 3rd. Int. Symposium DOA, Roma, 2001.
20. Falcarin, P. and Alonso, G.: Software Architecture Evolution through Dynamic AOP. European Workshop on Software Architecture (EWSA 2004), pp. 57-73, St. Andrews, UK, May (2004).
21. Ommering, R., Linden, F., Kramer, J. and Magee, J.: The Koala Component Model for Consumer Electronics Software. IEEE Computer, 33(3):78–85, March (2000).
22. ACME Home page, http://www-2.cs.cmu.edu/~acme/acme_downloads.html, (2005).