

Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform

Ackbar Joolia¹, Thais Batista², Geoff Coulson¹, Antonio Tadeu A. Gomes³

¹*Computing Department, Lancaster University, Infolab21, LA1 4WA, Lancaster, UK*

²*Departamento de Informática (DIMAp), Universidade Federal do Rio Grande do Norte (UFRN) 59072-970, Natal – RN, Brazil*

³*Laboratório Nacional de Computação Científica, (LNCC), 25651-075, Petrópolis – RJ, Brazil*

Abstract

Recent research has recognised the potential of coupling ADLs with underlying runtime environments to support systematic and integrated “specification-to-deployment” architectures. However, while some promising results have been obtained, much of this research has not considered the crucial issue of causally-connected dynamic reconfiguration and has considered only domain-specific areas. In this paper we discuss a specification-to-deployment architecture called *Plastik* that employs an extended general-purpose ADL and is underpinned by an efficient runtime that is suited both for high-level application development and low-level systems development (e.g. embedded systems). Runtime reconfiguration is supported both at the ADL level and at the runtime level, and both programmed reconfiguration and ad-hoc reconfiguration are supported. The paper focuses on the mapping of ADL-level specifications to runtime instantiations and on the necessary runtime support for causally-connected dynamic reconfiguration.

1. Introduction

Architecture Description Languages (ADLs) [9, 16,17,18,22] are aimed at the specification of high-level system architectures, described in terms of components and connectors. Recent research [2,5,10,20,26] has recognised the potential of coupling ADLs with underlying runtime environments to support systematic and integrated *specification-to-deployment* architectures. However, most of these architectures are *static* both at the ADL level and at the level of the generated executable—i.e. principled dynamic reconfiguration is not supported. Furthermore, they have considered only particular application areas (e.g. consumer electronics). More generally, we believe that the state of the art still lacks a fully systematic approach to providing a smooth and causally-connected link between high-level

architecture specifications and efficient low-level implementations.

In this paper we discuss a specification-to-deployment architecture called *Plastik*. *Plastik* employs a general purpose ADL which is systematically mapped onto an underlying runtime component model. Runtime reconfiguration is supported both at the ADL level (in terms of modified specifications) and at the component model level (in terms of programmatic manipulation of runtime components). Furthermore, runtime reconfiguration is supported under two headings: *programmed* and *ad-hoc*. Programmed reconfiguration pertains to reconfiguration operations that have been foreseen at design time. It is supported at the ADL level in terms of ‘condition-action’ statements which compile into corresponding runtime representations. Ad-hoc reconfiguration, on the other hand, involves changes that were *not* foreseen at system design time. Ad-hoc reconfiguration is not specified at the ADL level; rather, the approach is to annotate ADL specifications with *invariants* that restrict the allowable range of permissible ad-hoc reconfigurations.

A previous publication [29] has provided a general overview of *Plastik*. In this paper we focus particularly on the mapping from the ADL level to the runtime level, and on the runtime itself. The remainder of the paper is structured as follows: Section 2 gives a brief overview of the *Plastik* architecture. Then, Sections 3 and 4 respectively provide background on our ADL extensions, and on the underlying runtime component model. Subsequently, Section 5 provides detail on the implementation of the system with particular emphasis on the runtime machinery, the ADL-to-runtime mapping, and the management of runtime reconfiguration. Finally, Section 6 discusses related work and Section 7 offers our conclusions.

2. The Plastik architecture

The Plastik architecture comprises three levels: the *style*, *system*, and *runtime* levels (see Figure 1). The *style level* contains generic ADL types that can be used in the specification of families of systems. More specifically, a style specifies a set of types of components and connectors together with invariants on how these can be combined. The *system level* then contains instances of specific styles that follow the invariants imposed by the style. Further, per-system invariants can be added. Finally, the *runtime level* contains the runtime machinery that supports the execution of a system and polices its reconfiguration.

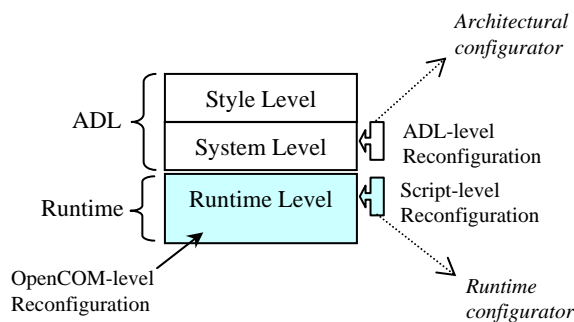


Figure 1. The Plastik architecture

At the style and system levels we employ an extended version of the ACME/Armani ADL [8]. Details of our extensions are given in Section 3. We have selected ACME/Armani because unlike many ADLs, it offers sufficient generality to straightforwardly describe a wide variety of types of systems. Also, it comes with tools that provide a good basis for the pragmatics of designing and manipulating architectural descriptions and generating code.

At the runtime level we employ our OpenCOM component model [12]. This has been widely applied in a range of application domains including real-time and embedded environments [11,19]. It natively supports a range of mechanisms to support runtime reconfiguration. Details are given in Section 4.

Plastik's support for programmed reconfiguration is realised in terms of the above-mentioned extensions to ACME/Armani. In terms of ad-hoc reconfiguration, Plastik supports this at three levels:

- **ADL-level reconfiguration** involves submitting a *reconfiguration specification*. This is an (extended) ACME/Armani specification that defines a delta over the currently running system. As explained later, the enactment of this

specification results in corresponding changes being effected at the runtime level.

- **Script-level reconfiguration** involves submitting a *reconfiguration script* that is written in terms of OpenCOM runtime API operations. In execution, Plastik ensures that the script does not violate any of the invariants that were expressed at either the style or system levels. Reconfiguration scripts can be generated from ADL-level reconfiguration specifications or they can be hand coded.
- **OpenCOM-level reconfiguration** involves making ad-hoc calls on the OpenCOM runtime API. Again, Plastik ensures that these calls do not violate any invariants.

Supporting reconfiguration at both the ADL and runtime (script and OpenCOM) levels raises issues of *causal connection*—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 script or OpenCOM-level ad-hoc reconfiguration may lead to rejection of a subsequent ad-hoc reconfiguration attempt at the ADL level due to an inconsistency having been introduced—e.g. the ADL-level reconfiguration request may refer to some component that has previously been removed by a prior runtime-level reconfiguration. In practice, we expect that most systems will employ either ADL-level or runtime-level ad-hoc reconfiguration but not both at the same time. Use of the runtime level is more appropriate in low-level system environments that are driven primarily by dynamic events in operating systems, for example. Use of the ADL-level, on the other hand, is more appropriate for higher system levels that are primarily driven by applications or GUIs.

3. The ADL level

A standard ACME specification is written in terms of *components*, *ports*, *connectors*, *roles*, *attachments*, *representations*, and *systems*. *Components* are (potentially composite) computational encapsulations with interfaces called *ports*. Ports are bound to ports on other components using first-class intermediaries called *connectors*. These support so-called *roles* that attach directly to ports by means of *attachments*. *Representations* are alternative decompositions of a given component; they reify the notion that a component may have alternative implementations. *Systems* are defined as graphs in which the nodes are components and the edges are connectors.

In addition, ACME specifications employ *properties*, *types*, and *architectural styles*. *Properties* are $\langle name, type, value \rangle$ triples that can be attached as annotations to any of the above-mentioned ACME elements (except attachments). *Types* are used to capture recurring structures and relationships. The ACME type system provides an additional dimension of flexibility by allowing ‘type extensions’ through the use of the *extended with* construct. *Architectural styles* define sets of types of components, connectors, and properties, together with rules that constrain how instances of types may be composed in a reusable architectural domain (cf. the Plastik style level).

Finally, *invariants* (or constraints) over ACME architectures are defined using the *Armani* extensions to ACME. Armani [28] is a FOPL-based sub-language that can be used to express invariants on system composition, behaviour, and properties. Invariants are comprised of standard logical connectives, existential and universal quantifiers, and Armani logical functions (both built-in and user-defined). Although Armani appears to introduce an element of dynamicity, it is important to emphasize that ACME/Armani does *not* support dynamic runtime reconfiguration of systems [9,17,29].

3.1 ACME language extensions

To enable ACME to express programmed reconfiguration we propose the following four additional constructs. These cover both *when* programmed reconfiguration should take place and *what* should be changed in any particular reconfiguration attempt.

- *on* ($\langle condition \rangle$) *do* $\langle actions \rangle$: this construct allows the ADL programmer to express runtime conditions (*condition* clause) under which programmed reconfigurations should take place, together with a specification of what should change (*actions* clause). The condition clause is an arbitrary Armani expression, and the actions clause contains arbitrary ACME/Armani statements. The intended semantics are that the elements specified in the actions clause are instantiated when the condition “triggers”. Note that triggering only occurs on a *transition* of the condition from false to true.
- *detach* $\langle role \rangle$ *from* $\langle port \rangle$; and *remove* $\langle elem \rangle$: these are mainly employed in *on-do* actions to enable (partial) architectures to be dismantled as well as constructed. *Detach* deletes an attachment between a role and a port; and *remove* destroys an existing component, connector or representation.

Removal of elements is only possible when they are no longer involved in an attachment.

- *dependencies* $\langle statements \rangle$: this construct allows the expression of runtime dependencies between architectural elements. In its *statements* clause it specifies an arbitrary set of ACME elements that are needed by the ‘target’ element to which the dependencies construct is attached. The semantic is that whenever the target element is instantiated (e.g. in the *actions* clause of the *on-do* statement) the stated dependencies are also instantiated; and whenever the target element is destroyed (using *remove/detach*) the stated dependencies are also destroyed.
- *active property* $\langle propertydefinition \rangle$: this is a special type of property that can only be attached to ports and roles. Its purpose is to refer to the dynamic flow of data over a port or role.

More details of these extensions can be found in the literature [26] and examples are given below.

3.2 Example

As OpenCOM has been widely exploited in programmable networking research [13], we employ a (highly simplified) *programmable networking* scenario (see Figure 2) to illustrate the extended ACME concepts and, later in Section 5.3, to illustrate the mapping from the ADL level to the runtime. The scenario involves an IP router that consists of packet *classifiers*, *forwarders* and *schedulers*. Classifiers analyse incoming packets and classify a packet based on fields in the packet header (source/destination IP address or port etc.); forwarders decide to which output link each packet will be transmitted; and schedulers insert packets into specific queues and choose from among queued packets which one to transmit next on each output link.

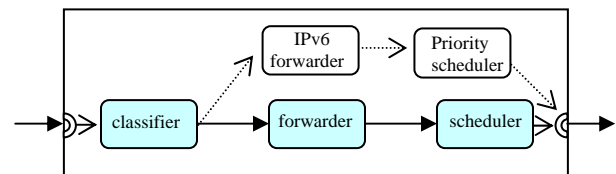


Figure 2. A basic router architecture

Figure 3 shows a Plastik style that describes the classifier, forwarder and scheduler types shown in

Figure 2 (plus a suitable connector type)¹. The invariant at the bottom defines the set of valid topologies for routers: connections are only allowed between classifiers, forwarders and schedulers, and no cycles are permissible. Note that the specification employs an ‘active property’ that supports runtime monitoring of the packets flowing through the associated port. The active property statement is shown in bold font (the same convention is also used in Figure 4 in which the other extensions are exemplified).

```

style Router = {
  property type IPPacket = record
    [ptype: string; IPHdr: string;
                                     Payld: string];

  component type Classifier: PlastikMF = {
    ProvidedPort classify = {
      property packet: IPPacket;
    };
    RequiredPort fromcls = {
      active property packet: IPPacket;
    };
  };

  component type Forwarder: PlastikMF = {
    ProvidedPort forward;
    RequiredPort fromfwd;
  };

  component type Scheduler: PlastikMF = {
    ProvidedPort store;
    RequiredPort fromsch;

    property algorithm: string;
    property maxQueueLength: int;
  };

  connector type connPath: PlastikMF = {
    ProvidedRole source;
    RequiredRole sink;
  };

  // only allow connections between
  // classifiers, forwarders and schedulers
  invariant ValidTopology(self);

  function ValidTopology(sys:system): boolean =
    forall c1,c2:component in sys.components |
      (reachable(c1,c2) =>
        ((satisfiesType(c1, Classifier) and
          satisfiesType(c2, Forwarder))
         or
          (satisfiesType(c1, Forwarder) and
           satisfiesType(c2, Scheduler)))
        and
        (!(satisfiesType(c1,Classifier) and
           satisfiesType(c2, Scheduler))))
      and
      // no cycles allowed...
      !reachable(c1,c1);

}; // end style

```

¹ All style definitions inherit from a super-style called *PlastikMF*. This contains, for example, the *ProvidedPort* and *RequiredPort* port definitions, and the *ProvidedRole* and *RequiredRole* role definitions.

Figure 3. A router style description

Figure 4 illustrates an example system that conforms to the Router style defined in Figure 3. This system-level architecture contains specific instances of the classifier, forwarder and scheduler types which are connected using two connectors.

Note that the system definition includes an *on-do* statement, and associated *detach*, *remove*, and *dependencies* statements. The collective effect of these is that on detecting an IPv6 packet in a stream of IPv4 packets, the configuration should switch over to subsequently supporting IPv6 instead of IPv4. The detection of an IPv6 packet is performed by the active property defined in Figure 3.

```

system RouterInst : Router, PlastikMF = {
  component cls: Classifier;
  component fwd: Forwarder;
  component sch: Scheduler;
  connector CtoF: connPath;
  connector FtoS: connPath;

  attachments {
    cls.fromcls to CtoF.source;
    CtoF.sink to fwd.forward;
    fwd.fromfwd to FtoS.source;
    FtoS.sink to sch.store;
  };

  on (cls.fromcls.packet.ptype = 'IPv6') do {
    detach CtoF.sink from fwd.forward;
    detach FtoS.sink from sch.store;
    remove fwd;
    remove sch;
    component IPv6Fwd: Forwarder = {
      dependencies {
        // component IPv6 depends on a scheduler
        // implementing a Priority algorithm
        component PriScd: Scheduler extended with
        {
          property algorithm: string = "Priority";
        };
        connector IPv6CtoF,IPv6FtoS: connPath;

        attachments {
          cls.fromcls to IPv6CtoF.source;
          IPv6CtoF.sink to Ipv6Fwd.forward;
          IPv6Fwd.fromfwd to IPv6FtoS.source;
          IPv6FtoS.sink to PriScd.store;
        };
      };
    };
  };
};

```

Figure 4. A router system

We give examples using these specifications in Section 5.3.

4. Background on OpenCOM

OpenCOM [14] is a small and efficient component-based programming technology that is language

independent, and intended to support the development of low-level systems as well as applications, and to support their runtime reconfiguration. For example, we have used it to construct adaptive middleware platforms and active networking environments [11]. In this section we provide only a brief overview of OpenCOM; more detail is available in the literature.

4.1 The OpenCOM programming model

Figure 4 gives a high-level view of the elements of OpenCOM’s component-based programming model, and Figure 5 shows (a simplified version of) the *component runtime kernel* (CRTK) API that supports this model. The API also incorporates a built-in transaction service that allows a sequence of reconfiguration-related operations (e.g. *load()*, *bind()*, *destroy()* etc.) to be executed atomically, with rollback on failure. To support this, the CRTK adds a transaction ID argument to each API call which, for clarity, is not shown in Figure 5.

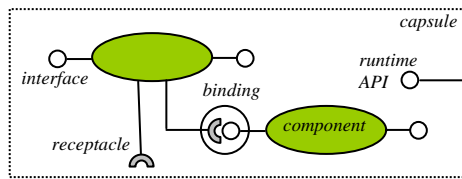


Figure 4. The OpenCOM component model

```

template load (comp_type name, predicate p);
comp_inst instantiate (template t);
status unload (template t);
status destroy (comp_inst comp);
comp_inst bind (ipnt_inst interface,
               ipnt_inst receptacle);
status putprop (ID entity, ID key,
               opaque value);
opaque getprop (ID entity, ID key);
status notify(Callback *callback);

```

Figure 5. OpenCOM’s CRTK API

Capsules are containing entities that offer the CRTK API and define a name space for contained components. *Components* are language-independent encapsulated units of functionality and deployment that interact with other components exclusively through “interfaces” and “receptacles” (see below). Component templates are deployed into capsules using *load()* and instantiated using *instantiate()*. Templates are loaded from a repository and are annotated with $\langle key, value \rangle$ pairs that can be used as a basis of a predication for template selection. Component templates are garbage collected using *unload()*, and instances are destroyed using *destroy()*.

Interfaces are sets of operation signatures and associated datatypes. Components can bear multiple interfaces. *Receptacles* are “required” interfaces used to make explicit the dependencies of one component on others. A *binding* is an association between an interface and a receptacle; bindings are created using *bind()* which returns a component instance representing the binding (thus bindings are deleted using *destroy()*). *Putprop()* and *getprop()* give access to an internal CRTK *registry*, which records arbitrary meta-data, expressed in terms of $\langle key, value \rangle$ pairs, that can be attached at runtime to any component model element.

Finally, the purpose of *notify()* is to inform interested parties of activity on the API. When a callback is registered with *notify()*, every subsequent call on the CRTK (i.e. of *load()*, *bind()* etc.) is reported to the callback. The callback can furthermore prevent the attempted call from going ahead by returning a value of *false*. We refer to *notify()* again in Section 5.1.

4.2 Reflective meta-models

Above the CRTK, OpenCOM supports additional generic services that facilitate the construction of complex systems and applications. These are themselves implemented in terms of components and are thus optional in any given capsule configuration. Key among these is a set of *reflective meta-models* [14] that facilitate dynamic reconfiguration of systems by permitting different system aspects to be programmatically inspected, adapted and extended at runtime. The following reflective meta-models are of most relevance to this paper: The *architecture meta-model* exposes the compositional topology of the components in a capsule in terms of a causally-connected graph structure; the *interface meta-model* allows one to discover information about interface types at runtime and to invoke interface instances that are dynamically discovered at runtime; and the *interception meta-model* allows one to interpose interceptors at bindings between component interfaces. In addition, we rely on aspects of the *resources meta-model* to freeze threads so that components can be safely replaced at runtime.

4.3 Component frameworks

Above the granularity of individual components, a key pattern employed in OpenCOM is to construct applications or systems in terms of *component frameworks* (CFs). CFs are tightly-coupled sets of components that work together to address some

focused area of functionality. Crucially, CFs accept ‘plug-in’ components, deployed at runtime, which modify or extend the CF’s behaviour. They also impose constraints on their plug-ins to prevent nonsensical compositions. As an example, a “protocol stack” CF might accept protocol components as plug-ins, and constrain these plug-ins to be composed linearly.

4.4 ACME-to-OpenCOM mapping

The basic mapping from ACME concepts to OpenCOM concepts is shown in Table 1. The system-to-CF correspondence is central: ADL-specified styles naturally correspond to CFs as domain-specific units of re-usable and dynamically reconfigurable functionality. We also assume a one-to-one mapping between components in the ADL definition and the OpenCOM components that underpin them. This simplifies causal connection at runtime between the ADL and runtime layers. Most of the rest of the correspondences are intuitively clear. We choose to map connectors to OpenCOM components because this enables us to easily support ‘abstract’ connectors such as SQL links or RPC channels. Similarly, we map ACME attachments to OpenCOM bindings because both represent associations between instances of interface requirement and provision.

Table 1. ACME-to-OpenCOM correspondence

ACME/Armani	OpenCOM
<i>system</i>	<i>CF</i>
<i>component/ connector</i>	<i>component</i>
<i>port/ role</i>	<i>interface/ receptacle</i>
<i>attachment</i>	<i>binding</i>
<i>representation</i>	<i>composite component</i>
<i>property</i>	<i>CRTK registry meta-data</i>

5. The runtime infrastructure

The runtime infrastructure required to support an OpenCOM CF that has been generated from an ADL-level system specification consists primarily of the following elements (see Figures 1 and 7):

- **Architectural configurator** This compiles specifications (e.g. the specification given in Figures 3 and 4), written in the extended ADL, into *configuration* and *reconfiguration scripts*. It is required at runtime because it needs to deal on the fly with ADL-level reconfiguration requests that modify existing specifications.

- **Runtime configurator** This per-CF element executes reconfiguration scripts. It is also responsible for managing the runtime representations of ADL-specified invariants, and for policing script-level and OpenCOM-level ad-hoc reconfiguration requests based on dynamic evaluation of these invariants.
- **Proxy CRTK** This is a per-CF version of the per-capsule CRTK. Each CF is given a dedicated proxy CRTK to provide inter-CF isolation. Components in one CF are prevented from interfering with components in other CFs because they have access neither to the proxy CRTKs of other CFs, nor to the per-capsule CRTK itself. Note, however, that components in different CFs can communicate directly as long as this doesn’t violate any invariants imposed by the CFs.
- **Lua interpreter** The above-mentioned scripts, as well as the runtime representations of ADL-level invariants and *on-do* conditions and actions, are all uniformly implemented as programs in the interpreted programming language Lua [21]. We therefore require a Lua interpreter in each capsule.
- **OpenCOM reflective meta-models** These are used in evaluating invariants and *on-do* conditions as discussed below.

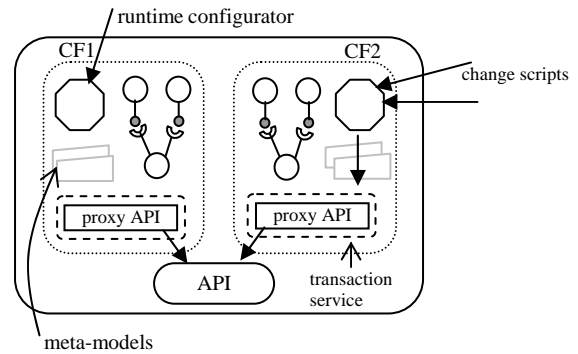


Figure 7. Runtime infrastructure

In Section 5.1 we focus on the runtime configurator and then, in Section 5.2, on supporting the extended ADL constructs.

5.1 The runtime configurator

The first task of the runtime configurator is to initialise the CF by executing its configuration script to establish the initial state of the CF in terms of loading, instantiating and binding the necessary OpenCOM components, and ensuring that the rest of the runtime machinery is instantiated.

Having established the runtime environment, the configurator installs a set of *predicates*. These are derived from two sources: *i*) the condition clauses of *on-do* statements, and *ii*) standard ACME/Armani invariants. The evaluation of these predicates takes place in the context of a callback that has been registered with the proxy CRTK's *notify()* operation (see Section 3). If an *on-do* condition predicate evaluates to *true*, the corresponding *on-do actions* script is executed. On the other hand, if a standard ACME/Armani-derived predicate evaluates to *false*, the transaction in which the predicate evaluation is embedded is aborted and rolled back.

As mentioned, predicates are uniformly represented as Lua scripts that are automatically generated from the ADL level and evaluated by the Lua interpreter. Ordinary (i.e. non-active) ADL-level properties are “grounded” in *<key, value>* pairs held in the CRTK registry. These are read and potentially modified both by script execution and by application code. As an example of the latter, consider the case of the *maxQueueLength* property attached to the scheduler in Figure 3. The scheduler component would be expected to manually update the value of the corresponding registry entry whenever the queue length changes. *Active* ADL-level properties, on the other hand, are grounded in OpenCOM interceptors attached to the binding component that realises the port or role to which the active property is attached. An example of an active property, *packet*, is given in Figure 3. Currently, we require active property interceptors to be hand-coded, but we are looking at the extent to which they can be automatically generated.

Finally, as well as registry lookup and interceptor evaluation, predicate evaluation also relies on consulting the reflective meta-models. In particular, the interface meta-model is consulted on dynamic typing issues, and the architecture meta-model on topological issues (e.g. if a predicate asks if a given component is bound to some other component, or to implement the *reachable()* function shown in Figure 3).

5.2 Supporting the extended ADL constructs

We have already seen how the *on-do* construct is supported in terms of condition predicates and action scripts, and how active properties are represented as interceptors. We now briefly discuss the support of the other constructs.

First, the *detach* and *remove* constructs are straightforwardly implemented in terms of the CRTK-level *destroy()* operation. The fact that a CRTK-level operation is used automatically guarantees that these

statements are successfully executed only if they do not violate any invariant. Second, the *dependencies* statement is handled similarly to the *on-do* statement: it simply maps to a sub-script whose execution is ‘guarded’ by a predicate that evaluates to *true* whenever the element to which the statement is attached is instantiated or deleted (as appropriate).

5.3 Examples

To illustrate how ADL specifications are mapped to their runtime representations, we now revisit the IP router example (Figures 3 and 4), described in Section 3.2. In particular, Figure 8 shows the Lua derivation of the *ValidTopology()* invariant in the router style. Functions prefixed by *crtk* and *amm* (shown in bold font) are mapped by the Lua interpreter onto one or more operation invocations on the CRTK API and the reflective meta-models respectively. Note that calls to these include a transaction identifier; it is assumed that this has been obtained by the caller.

```
function lps_ValidTopology(tid)
  local forAllExp_1 = true
  for _,c1 in ipairs(amm.enumInstances(tid))
    local forAllExp_2 = true
    for _,c2
      in ipairs(amm.enumInstances(tid))
        local boolExp = true
        if amm.reachable(c1, c2) then
          local c1cat = crtk.getProp(tid,
            c1, "Category")
          local c2cat = crtk.getProp(tid,
            c2, "Category")
          boolExp =
            ((c1cat == "Classifier" and
              c2cat == "Forwarder") or
             (c1cat == "Forwarder" and
              c2cat == "Scheduler")) and
            (not (c1cat == "Classifier" and
                  c2cat == "Scheduler")) and
            not amm.reachable(tid, c1, c1)
        end --if
        if not boolExp then
          forAllExp_2 = false; break
        end --if
      end --for
    end --for
  if not forAllExp_2 then
    forAllExp_1 = false; break
  end --if
end --for
return forAllExp_1
end --function
```

Figure 8. Lua predicate script

Configuration and reconfiguration specifications in ACME/Armani are translated into Lua scripts in much the same way as depicted in Figure 8: specific Lua functions map to operations on the CRTK API.

As a second example, consider the implementation of the active property instance in Figures 3 and 4. This is implemented as an interceptor attached to the binding between the classifier and the forwarder components, which inspects the header of each packet. When it detects an IPv6 packet, the interceptor sets a corresponding property in the CRTK registry. This, in turn, triggers a call of the *notify()* callback situated in the runtime configurator which, on evaluating its predicates, discovers that the condition guarding the actions part of the *on-do* statement has triggered and therefore executes the *on-do* actions clause in a new transaction. The *dependencies* statement within the *on-do* statement is handled similarly: as soon as the IPv6 forwarder is instantiated, this triggers an (implicit) guard that enables the execution of a sub-script containing Lua equivalents of the statements inside the *dependencies* statement.

6. Related work

Aster [7,15] is a development environment that aims to ease the construction of distributed systems on top of object-oriented middleware. It provides an ADL with an embedded property sublanguage that supports the architectural description of distributed applications and their non-functional requirements. A tool selects appropriate middleware objects based on an application description, and integrates them with the application through the generation of appropriate proxies. As regards (re)configuration, Aster shares similarities with Plastik, including support for ad-hoc reconfiguration; however, Aster does not support programmed reconfiguration. Furthermore, Aster is not a general purpose approach; for example, it cannot realistically be used for low level applications such as embedded systems.

Koala [26] is a component model that uses an ADL based on Darwin [16] to manage the complexity of software in consumer electronics products. However, dynamic reconfiguration is restricted to switching between pre-existing components based on statically defined conditions. Furthermore, there is no causal connection between the ADL level and the implementation.

Knit [3] is a component definition and linking language that promotes the wrapping and reuse of existing code (in C). One of its main features is the minimisation of componentisation overhead; thus it is particularly useful for designing and implementing complex, low-level systems. Knit also provides some support for formally defining and enforcing architectural constraints, but it has nothing comparable

to a fully general ADL. Also, in comparison with Plastik, Knit is programming language-specific and does not support either ad-hoc or programmed reconfiguration.

FORMAware [23] is a reflective component-based framework that augments explicit architectural description with meta-information to constrain reconfiguration. To avoid inconsistency it checks architectural constraints according to “style rules” that restrict the types of architecture elements in use and possible configurations. A transaction service manages reconfigurations. However, programmed reconfiguration is not supported. Another fundamental difference between FORMAware and Plastik is that we adopt a formal approach to the definition of constraints, whereas FORMAware style rules are described as pieces of procedural code. This lowers its level of abstraction with respect to Plastik.

Fractal [10] is a hierarchically-structured Java-based component model which uses an XML-based ADL to specify the high level structure of an application, and which provides runtime reflective features to support dynamic reconfiguration. However, the ADL level, like FORMAware and Koala, has no formal support for the description of constraints and no causal connection between the ADL and implementation levels.

Mae (Managing Architectural Evolution) [27] is an architectural evolution environment that uses xADL [30] to specify architectures. While it explicitly supports reconfiguration, this is achieved through completely different means to those adopted by Plastik. In particular, Mae employs a versioning mechanism combined with a check-out/check-in approach. Other key differences are that Mae supports only programmed reconfiguration (architectural configurations are selected from a ‘version space’), and again lacks a formal approach with which to impose constraints upon reconfiguration.

The work discussed in [27] is based on the C2 ADL [25]. It presents an approach to runtime software evolution that places emphasis on connectors being represented in the implementation and playing a central role in supporting evolution. Runtime changes are applied to an intermediate representation of the architecture model, which is present at runtime. This work has similarities to Plastik; however, changes can only be applied at the architectural model level (whereas Plastik additionally supports script-level and component-model level reconfiguration). Furthermore, constraint on reconfiguration is focused on connectors, whereas Plastik (thanks to ACME/Armani) supports a fully general constraint mechanism that can involve any architectural element.

Finally, ArchWare [24] is similar to Plastik in being driven by an ADL with formal support and offering implementation level reconfiguration via reflection. For the latter purpose, ArchWare uses an active execution graph with a programmable interface as a representation of the executing system. In contrast, Plastik adopts an efficient component runtime as its execution element and focuses on the mapping from an (extended) ADL to this runtime.

7. Conclusions

In this paper, we have proposed a causally-connected specification-to-deployment architecture called Plastik which can be used for both application and low-level systems development. Plastik's ADL-level adds four new constructs to ACME/Armani that, together with runtime evaluation of Armani invariants, support the expression and management of both programmed and ad-hoc reconfiguration.

The main focus of this paper has been to illustrate the mapping from an ADL-level description of a system to an OpenCOM component framework, and to explain how programmed and ad-hoc reconfigurations are supported at runtime in terms of predicate management and transactional reconfiguration management.

Although we have not yet investigated this in detail, we believe that Plastik systems can easily achieve high performance. This is partly because of the already-validated performance of OpenCOM [12], and partly because OpenCOM allows Plastik to run in such a way that its runtime reconfiguration facilities operate in an 'out-of-band' manner. That is, they only incur an overhead when the CRTK is called to load/ unload/ bind/ unbind etc. In the normal 'in-band' execution of an application in a stable state they incur zero overhead. A slight exception to this is the use of interceptors to underpin active properties: developers should take care not to over-use this facility where performance is an issue.

Currently, we are using the AcmeLIB tools [1] to implement compilation and predicate generation, and the Lua interpreted language for the expression and evaluation of predicates. We have successfully trialled key aspects of the design, including:

- the runtime configurator: this is integrated with the Lua interpreter and the CRTK API, and a Lua-to-OpenCOM interface allows scripts to access the CRTK API and the meta-models;
- on-the-fly predicate management: invariants/ conditions can be modified at the ADL level and the corresponding predicates updated at the

runtime level without recompilation or even disruption of the system;

- basic transactional support: a simple, deferred update-based, transaction mechanism has been implemented.

However, we do not yet have a fully working compiler that generates Lua from ACME.

Future work will involve several areas. First, we want to add support for the replacement of stateful components by means of specification- and mechanism-level support for state transfer. Second, we want to optimise predicate evaluation. Currently, we perform redundant evaluations of predicates that could not possibly be affected by the event triggering the evaluation. We plan to optimise this with the aid of the constraint tree approach developed in [6]. Third, we want to investigate how predicates can be represented and managed in primitive systems for which the memory overhead of Lua cannot be supported. This is of central interest to us as we are investigating in related research the use of Plastik in highly restrictive environments such as embedded systems and sensor nodes.

More generally, we also plan to extend the upper layers of Plastik to accommodate a wider range of specification languages. We have already used a DSL-oriented specification framework [4] for this purpose, and are also starting to investigate the use of UML [19].

8. References

- [1] ACME Home page, http://www-2.cs.cmu.edu/~acme/acme_downloads.html, (2005).
- [2] A. Ramdane-Cherif, and N. Levy, "An Approach for Dynamic Reconfigurable Software Architectures", Integrated Design and Process Technology, IDPT-2002, June (2002).
- [3] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide, "Knit: Component Composition for Systems Software", In *Proceedings of 4th Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego (CA), USA, Oct. 2000, pp. 347—360.
- [4] A.T.A. Gomes, G. Coulson, G.S. Blair, and L.F.G. Soares, "A component-based approach to the creation and deployment of network services in the programmable Internet". Technical Report MCC-42/03, PUC-Rio, Brazil, 2003. Available at <http://www.inf.puc-rio.br/>.
- [5] A. Van der Hoek, D. Heimbigner, and A. Wolf, "Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois", Technical Report CU-CS-849-98, University of Colorado, (1998).
- [6] C. Efstathiou, A. Friday, N. Davies, K. Cheverst, "Utilising the Event Calculus for Policy Driven

- Adaptation on Mobile Systems”, In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, Monterey, Ca., U.S., J. Lobo, B. J. Michael and N. Duray (eds.), IEEE Computer Society, pp. 13-24, June, 2002.
- [7] C. Kloukinas, and V. Issarny, “Automating the Composition of Middleware Configurations”, In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE’2000)*, Grenoble, France, Sept 2000, pp. 241—244.
- [8] D. Garlan, R. Monroe, and D. Wile, “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).
- [9] D. Wile, “Using Dynamic Acme”, In *Proceedings of a Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, December, (2001).
- [10] E. Bruneton, T. Coupaye, and J-B. Stefani, “Recursive and Dynamic Software Composition with Sharing”, Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain, (2002).
- [11] G. Blair, G. Coulson, and P. Grace, “Research Directions in Reflective Middleware: the Lancaster Experience”, *Proc. 3rd Workshop on Reflective and Adaptive Middleware (RM2004)* co-located with Middleware 2004, Toronto, Ontario – Canada, Monday, October 18th 2004.
- [12] G. Coulson, G.S. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama, “OpenCOM v2: A Component Model for Building Systems Software”, *Proceedings of IASTED Software Engineering and Applications (SEA’04)*, Cambridge, MA, ESA, Nov (2004).
- [13] G. Coulson, G.S. Blair, D. Hutchison, A. Joolia, K. Lee, J. Ueyama, A.T. Gomes, and Y. Ye, “NETKIT: A Software Component-Based Approach to Programmable Networking”, *ACM SIGCOMM Computer Communications Review (CCR)*, Vol 33, No 5, pp 55-66, October (2003).
- [14] G. Coulson, G.S. Blair, M. Clarke, and N. Parlavantzas, “The Design of a Highly Configurable and Reconfigurable Middleware Platform”, *ACM Distributed Computing Journal*, Vol 15, No 2, pp 109-126, April (2002).
- [15] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras, “The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms”, In *Proceedings of the IFIP/ACM International Conference on Middleware (Middleware’2000)*, Hudson River Valley (NY), USA, Apr. 2000.
- [16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, “Specifying Distributed Software Architectures”, In *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, pp. 137-153, September (1995).
- [17] N. Medvidovic, and R. N. Taylor, “A classification and comparison framework for software architecture description languages,” *IEEE Trans. Software. Eng.*, vol. 26, no. 1, pp. 70-93, 2000.
- [18] M. Shaw, and D. Garlan, “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, (1996)
- [19] P. Costa, G. Coulson, C. Mascolo, G.P. Picco, and S. Zachariadis, “The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems”, to appear 16th Annual International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC05), Berlin, Germany, 2005.
- [20] P. Falcarin, and G. Alonso, “Software Architecture Evolution through Dynamic AOP”, European Workshop on Software Architecture (EWSA 2004), pp. 57-73, St. Andrews, UK, May (2004).
- [21] R. Jerusalimsky, L. H. Figueiredo, and W. Celes, “Lua – an extensible extension language”, *Software: Practice and Experience*, 26(6):635-652, (1996).
- [22] R. J. Allen, R. Douence, and D. Garlan, “Specifying and Analyzing Dynamic Software Architecture”, In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE’98)*, March (1998).
- [23] R. Moreira, G. Blair, and E. Carrapatoso, “FORMAware: Framework of Reflective Components for Managing Architecture Adaptation”, 3rd. Int. Symposium DOA, Roma, 2001.
- [24] R. Morrison, et al, “Support for Evolving Software Architectures in the ArchWare ADL”, *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Oslo, Norway, 2004.
- [25] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. “A Component- and message-based architectural style for GUI software.”, *IEEE Transactions on Software Engineering*, pp 390-406, June 1996.
- [26] R. Ommering, F. Linden, J. Kramerand, and J. Magee, “The Koala Component Model for Consumer Electronics Software”. *IEEE Computer*, 33(3):78–85, March (2000).
- [27] R. Roshandel, A. Van der Hoek, M. Mikic-Rakic, and N. Medvidovic, “Mae – A System Model and Environment for Managing Architectural Evolution”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3 (2):240-276, (2004).
- [28] R. T. Monroe, “Capturing Software Architecture Design Expertise with Armani”, Technical Report CMU-CS-98-163, Carnegie Mellon University.
- [29] T. Batista, A. Joolia, and G. Coulson, “Managing Dynamic Reconfiguration in Component-based Systems”, to appear in EWSA 2005 2nd European Workshop on Software Architectures.
- [30] xADL Home Page:
<http://www.isr.uci.edu/projects/xarchuci/> (2005)