A component-based approach to the creation and deployment of network services in the programmable Internet[1]

Antônio Tadeu Azevedo Gomes[2]
atagomes@inf.puc-rio.br

Geoff Coulson
Computing Department
Lancaster University
Lancaster LA1 4YR, UK
g.coulson@lancaster.ac.uk

Gordon Shaw Blair
Computing Department
Lancaster University
Lancaster LA1 4YR, UK
g.blair@lancaster.ac.uk

Luiz Fernando Gomes Soares
lfgs@inf.puc-rio.br

PUC-RioInf.MCC42/03 November, 2003

Abstract: Dynamism and simplicity in service *creation* and service *deployment* are recurring preoccupations to service designers. Although significant research has been carried out in both areas, we believe there remains a need for a better integration of them. The present work is an effort to apply component-based software concepts 'anywhere', from high-level service specifications to low-level software implementation in programmable network devices. This paper presents LindaX, an architecture description language that aims at guiding designers through the use of these concepts in a principled way.

Keywords: telecommunication service engineering discipline; programmable networks; component software; frameworks; architecture description languages; quality of service.

Resumo: Dinamismo e simplicidade na *criação* e *implantação* de serviços são preocupações recorrentes para projetistas de sistemas de telecomunicações. Apesar de vários trabalhos abordarem ambas as áreas, acreditamos que ainda há a necessidade de uma melhor integração entre elas. Este trabalho objetiva aplicar conceitos de software baseado em componentes da maneira mais abrangente possível, desde especificações de serviço de alto nível até a implementação de software básico em dispositivos de rede programáveis. Este artigo apresenta LindaX, uma linguagem de descrição de arquiteturas cujo objetivo é guiar os projetistas no uso desses conceitos de maneira organizada.

Palavras-chave: engenharia de serviços de telecomunicações; redes programáveis; software baseado em componentes; frameworks; linguagens de descrição de arquitetura, qualidade de serviço.

[1] Submitted to Journal of Computer Communications, Special Issue on Activated and Programmable Internet, 2003.
[2] Sponsored by CNPq/ CAPES, Brasil.

1. Introduction

Rapid and cheap deployment of new telecommunication services is essential if network operators are to maintain or expand their market shares. Such demand encompasses two different dimensions.

First, there is an increasing need for openness and programmability in the service offering infrastructure. Current research and development in the area of network processors seems to cope pretty well with the traditional trade-off "speed versus programmability" (see [1, 25, 26] for examples). Based on results from that field, *programmable network architectures* abound. Independent of specific approaches for designing and developing programmable networks (see [10] for one possible classification), the major trends have been towards: (i) increasingly dissociating network software from network hardware and (ii) deploying IP-based multi-service networks.

The second dimension derives from the first one in the sense that concepts, principles and rules from the software engineering area can be applied in the realm of programmable network architectures to help in organising the process of service creation. More specifically, the inherent complexity of programmable networking software can benefit from the application of *well-established methods, techniques and tools* throughout the phases of service specification, design, implementation, verification and validation. Znaty & Hubaux [61] go farther in that direction by justifying the creation of a telecommunication service engineering discipline, which brings common software engineering concepts together with telecommunication specific requirements such as security, communication management, etc.

Although the idea of programmable networking is not new, and there has already been significant research in support of the service creation process, we believe there remains a need for a better integration of the two aforementioned dimensions. The approach we have adopted to achieve such integration is to apply the notions of *components*, *frameworks* and *architectural descriptions* in the realms of both programmable networking and service creation. Unlike other proposals, which advocate component-based approaches to service offering and service creation environments but only address specific concerns, we envisage components being uniformly applied at all levels, ranging from high-level service specifications to low-level packet processing implementation.

The approach presented in this paper, coming from a collaboration between the Catholic University of Rio de Janeiro (PUC-Rio) and Lancaster University, tries to deal with the dimensions of service offering and service creation by relating two independent component models: the *abstract model*, developed at PUC-Rio [14], which focuses on high-level concepts related to service specification and design; and the *concrete model*, developed at Lancaster [13], which is aimed at composing flexible software systems. This paper presents an *architecture description language* (ADL[3]) tailored to specifying adaptable communication systems – called *LindaX*[4] – that is derived from the abstract model. The structure of LindaX is centred around the use of *architectural styles* [42] as a means of formally describing architectural configurations of communication systems and their points of adaptation. LindaX has been developed in a modular way so that it can be independently extended to: (i) comprise different techniques for formal reasoning of architectural configurations, and (ii) support the synthesis of these configurations in diverse service offering environments. As an example of the latter, the present paper shows how LindaX can serve as a guideline for the application of a *component-based programmable networking toolkit* that is derived from Lancaster's concrete model. It is also demonstrated in this paper, by means of a detailed example, how associated tools can provide the necessary mapping between the abstract and concrete models by linking the architectural reasoning support given by LindaX to the features of extensible configuration management embedded within the component-based toolkit.

The paper is structured as follows. Section 2 gives an overview of the abstract model and introduces the LindaX type system. Section 3 presents the case for

[3] As stated by Medvidovic & Taylor [40], there is little consensus in the research community on what is an ADL and which level of support an ADL should provide to designers. However, it is reasonably accepted in the literature [2, 17, 19, 32,, 37, 39], as a minimum, that ADLs must explicitly model *computation* and *communication* entities (typically referred to as components and connectors), as well as their *configurations*. Moreover, it is usually argued that for the architectural descriptions to be of any use, an ADL must provide some *design tools* (e.g. to support architectural and/ or behavioural reasoning). In our belief, the language proposed in this paper fits in the set above.

[4] Pronounced /*'Lin-dush*/.

frameworks as a means of expressing adaptation constraints in the abstract model and illustrates how LindaX makes use of architectural styles to represent these constraints in a formal way. Sections 4 and 5 outline the main concepts underlying the concrete model and its support for reflective and framework-based constraint enforcement during adaptations. The mapping of the abstract notion of frameworks in LindaX onto component frameworks in the concrete model is also shown in Section 5. Following this, Section 6 presents the architectural configuration support in LindaX, and gives a detailed example of how the semantics of a configuration description in LindaX rely on associated architectural styles. Section 7 looks into related work both in the area of programmable networking and service creation. Finally, Section 8 is reserved to some concluding remarks and topics for further development.

2. The abstract model

The abstract view of the service creation and offering dimensions presented in this paper is extensively based on the *Service Composition Model* (SCM) [14]. SCM was initially targeted at the assessment and comparison of a comprehensive set of adaptability and programmability approaches found in the realm of communication systems, ranging from innovative solutions in programmable networking to legacy and standardised systems, such as ATM and MPLS. This section outlines the model concepts by means of two different views – the *architectural* and *execution* views – and introduces the extensible type system of the proposed LindaX ADL, which is derived from the model.

2.1. Architectural view

SCM provides a basic 'architectural vocabulary' for service creation, defining the concepts of *user components*, *service providers*, *ports*, *access points* and *attachments*, as depicted in Fig. 1.

A user component represents part of a communication system. User components have ports through which they communicate with other user components to make up the whole system. A service provider represents a generic communication infrastructure for the system. Service providers offer access points to which user components must attach their ports in order to communicate. These attachments delineate the SCM abstract view of a service offering environment. From the perspective of SCM, a service offering environment can be seen as an address space. However, it is not restricted to any particular kind of system. For instance, in SCM the abstraction of a service offering environment may represent anything from a single process (whose user components are modules or objects) to an internetworking system (whose user components are applications or protocol entities). Moreover, the architectural view is not restricted to representing software entities; user components and service providers may also represent hardware entities (e.g. network cards, physical links).

Both user components and service providers can be composed of other components and providers. However, components and providers representing

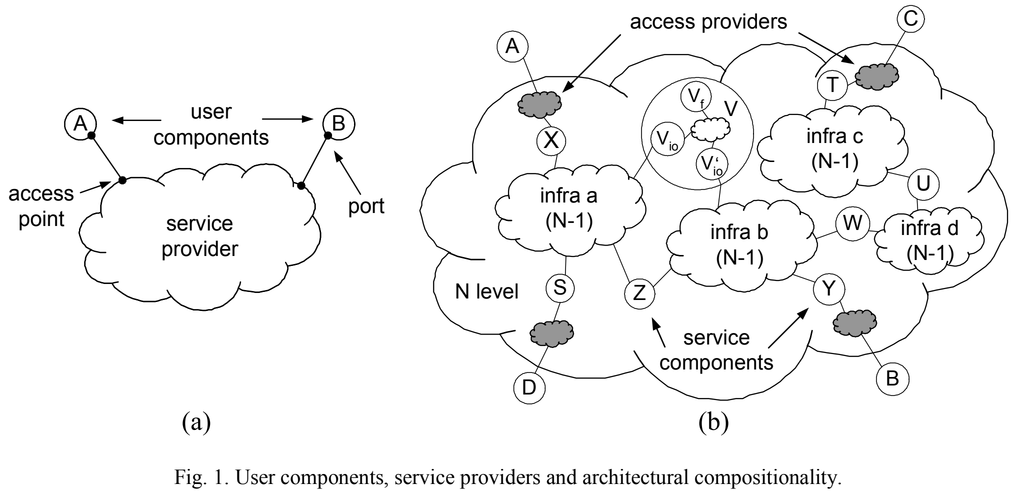


Fig. 1. User components, service providers and architectural compositionality.

purely hardware entities are always assumed to be primitive. Some kinds of compositions are illustrated in Fig. 1(b), which exemplifies one possible expansion of the service provider in Fig. 1(a).

Compound components implement their ports by exposing ports of internal components. Similarly, compound providers implement their access points by exposing access points of internal providers, the latter being called *access providers*. The other internal providers of a compound provider are named *infrastructure providers*, and its internal components are named *service components* (see Fig. 1(b)). Both the infrastructure and access providers can be structured so that the service components act as user components. Within these providers, other service components, infrastructure and access providers may show up. It is important to notice that the model does not prescribe any other particular semantics of composition, thus adhering to the requirements on component models introduced by van der Hoek [58]. In fact, SCM permits that composite elements have overlapping content, for example, as long as their realisation in a concrete model allows for (at least the emulation of) such sort of composition.

The nested organisation of user components and service providers allows modelling many different system patterns. For example, Fig. 1(b) shows an OSI-based architecture in which service components represent protocol entities implementing a compound provider (of level N) and communicating among them using infrastructure providers (of level N-1). In this architecture, user components make use of services by communicating with service components through access providers. However, SCM allows different system views, thus offering a means for designers to emphasise their points of interest. Taking again Fig. 1(b) as an example, instead of a vertical layered architecture a service designer could also think of a horizontal topological architecture in which access providers could represent wireless networks and infrastructure providers could represent the wired interconnecting backbone. The designer could also be more interested in the process architecture [51] to focus on the granularity of parallelism of a certain protocol suite. Thus, he or she could represent part of the system as depicted by the composite component *V* in Fig. 1(b), where the internal provider represents exactly the OSI concept of a local system environment (LSE).

The concepts of user components and service providers are essentially structural and semantically neutral in SCM; that is, no behaviour description structure is provided in [14]. Hence, user components and service providers manifest themselves only at run-time. The proposed LindaX ADL offers an extensible type system that allows associating additional information (e.g. expected behaviour) with SCM concepts at design-time.

The LindaX core type system consists of *component types*, *provider types* and *interface types*. The basic structure of these elements is shown in Fig. 2.

Each component type has a set of port descriptions. Likewise, each provider type has a set of access point descriptions. A component/ provider of a specific type must have *at least one* instance of each port/ access point described as part of the type. Each port and access point description has a set of *signatures* referring to specific interface types. This ultimately prescribes what types of interfaces must be implemented by a user component or service provider of a particular type.

The core type system is also semantically neutral, thus not restricting the kinds of computation or interaction patterns that can be described at design-time. LindaX interface types, for example, can describe both *provided* and *required* services. The latter are especially important to make explicit the dependencies of one user component or service provider on another, as will be further discussed in Section 4. Types in LindaX can be adorned with *attributes* describing some of the characteristics of the type. For example, an attribute *Cardinality* can be

```
InterfaceType intf1-name {...}
InterfaceType intf2-name {...}
ComponentType comp-name {
  Port port-name {
    Signature sig-name { Type = intf1-name }
    ... /* other signatures */
  }
  ... /* other port declarations */
}
ProviderType provider-name {
  AccessPoint ap-name {
    Signature sig-name { Type = intf2-name }
    ... /* other signatures */
  }
  ... /* other access point declarations */
}
```

Fig. 2. LindaX core type system.

attached to port and access point descriptions so as to constrain the number of their instances in a type declaration. Types can also be parameterised with regard to any internal attribute of the type specification. These features are particularly important for the sake of architectural reasoning and synthesis of systems, as will be illustrated in the following sections.

As an example of type definitions in LindaX, imagine that a designer wants to specify the composite component V in Fig. 1(b) as a packet forwarding entity (e.g. an IP module in a router). First of all, a set of provided/ required packet-passing interface types must be declared. It is supposed in the example that the designer opted for defining both push- and pull-oriented interface types. Then, the designer may declare types for components V_{io} and V'_{io} (representing input-output modules), each of them implementing exactly two packet-passing ports, and for component V_f (representing a forwarding module), which may implement at least two packet-passing ports. A type for the provider interconnecting these components must also be declared, with a variable number of packet-passing access points. Fig. 3 depicts a skeleton of these type definitions in (a much abbreviated notation of) LindaX. In the example it is assumed that, for configurations based on the aforementioned types (e.g. the composite component V), input-output modules are always passive entities (i.e. packets are pushed into them and later pulled from them) whereas forwarder modules are fully active (i.e. they pull packets from input-modules and later push the packets into output modules). The presentation of LindaX support for configuration descriptions based on its type system is deferred to Section 6.

```
InterfaceType PPacketPush {...}
InterfaceType RPacketPush {...}
InterfaceType PPacketPull {...}
InterfaceType RPacketPull {...}
ComponentType UIOModule {
  Port io {
    Cardinality = 2
    Signature in  { Type = PPacketPush }
    Signature out { Type = PPacketPull }
  }
}
ComponentType UForwardingModule {
  Port io {
    Cardinality = 2..
    Signature in  { Type = RPacketPull }
    Signature out { Type = RPacketPush }
  }
}
ProviderType SLocalEnv {
  AccessPoint io_src {
    Signature in  { Type = RPacketPull }
    Signature out { Type = RPacketPush }
  }
  AccessPoint io_dst {
    Signature in  { Type = PPacketPush }
    Signature out { Type = PPacketPull }
  }
}
```

Fig. 3. Example of type declarations in LindaX.

2.2. Execution view

The architectural view represents the run-time structure of a system, without regard to other aspects such as resource management and QoS provisioning. SCM provides two abstractions – *tasks* and *MediaPipes* – which relate the execution of computing and communication activities to resource partitions, over which QoS requirements (such as maximum communication delay, minimum acceptable processing quantum and period, etc.) can be defined and handled.

Tasks are associated with computations. Processes, threads and grid computations are examples of tasks. MediaPipes represent interactions between two or more components attached to a provider. Virtual connections, remote operation invocations and data within a single packet are examples of MediaPipes. More than one MediaPipe/ task may be related to the same provider/ component and one single MediaPipe/ task may span more than one provider/ component. For example, a possible execution view of the architecture depicted in Fig. 1 is show in Fig. 4, in which tasks A' and A'' are directly related to execution of user component A, whereas tasks YB' and YB'' span execution and communication activities involving components Y and B and the access provider between them. The relationships between elements of the architectural and execution views are further discussed in Section 2.3.1.

A MediaPipe/ task can be composed of other MediaPipes/ tasks, as for example in the case of grid computations and virtual connections. As in the

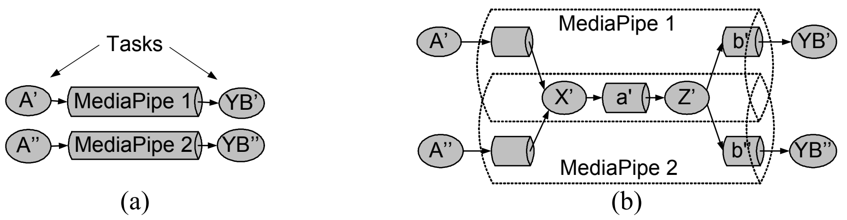


Fig. 4. MediaPipes, tasks and run-time compositions.

architectural view, SCM does not prescribe any additional semantics of composition in the execution view. For example, Fig. 4(b) sketches a scenario where MediaPipes *1* and *2* are composites that share the same internal tasks *X'* and *Z'* and MediaPipe *a'*. The internal MediaPipe could represent, for example, one virtual connection being multiplexed by two application-level flows, without any traffic classification within the connection.

2.3. SCM Metadata

SCM employs another important abstraction: the explicit representation of *metadata*. In SCM, metadata may comprise various types of information such as protocol stack configuration, the current network load, etc; in fact, everything that affects the process of service offering. To date, SCM defines two main metadata structure types: the *topology* and *resource* structures.

2.3.1. Topology structure

The topology structure represents the overall architecture of a running communication system, making explicit the relationship between the architectural and execution views. It consists of two compound graphs [55] called *architectural graph* (G_a) and *execution graph* (G_e), and a set of directed graphs called derivation graphs (G_d).

G_a and G_e are used to represent the hierarchical structure of the architectural and execution views, respectively. The vertices V_a of G_a are divided into two disjoint sets to distinguish vertices representing user components (V_{ac}) from service providers (V_{ap}). The same applies to the vertices V_e of G_e to make a distinction between tasks (V_{et}) and MediaPipes (V_{em}). Attachments are represented as edges in both graphs. The rules of composition presented in Sections 2.1 and 2.2 are included in the architectural and execution graphs as constraints on the configuration of these graphs. More specifically, if v is a composite vertex in G_a (G_e), then all edges traversing the boundary of v must emanate from a constituent vertex v_i such that v_i and v belong to the same set V_{ac} or V_{ap} (V_{et} or V_{em}). These constraints forbid the representation of illegal compositions, such as a composite component that implements a port by exposing an access point of a constituent provider, for example.

Each derivation graph G_{d_i} in the set G_d is used to represent the relationship between elements of the architectural and execution views. The vertices V_{d_i} of G_{d_i} are such that

$$\bigcup_i V_{d_i} = V_a \cup V_e \qquad (1)$$

An activity (whether it be a task or MediaPipe) being related to a component/ provider is represented by a directed edge in G_d. Again, rules of composition of the architectural and execution views are included in the derivation graphs as constraints on their configuration. Namely, vertices in V_{et} (V_{em}) may be linked either to a single vertex in V_{ac} (V_{ap}) or to a set of vertices in V_{ac} and V_{ap} – call it S. In the latter case, the architectural subgraph containing all vertices in S (and the edges between them) must be connected. These constraints hinder the representation of meaningless activities involving unrelated architectural elements.

2.3.2. Resource structure

The resource structure represents the sharing of resources among tasks and MediaPipes, thus permitting the association of QoS requirements with both types of activities. It consists of a forest of directed trees called *resource trees* (T_r).

Each resource tree T_r represents the sharing of a specific resource (or set of resources), such as CPU time, memory areas, link bandwidth or composite resources (see next paragraph), among tasks and MediaPipes. The vertices V_r of T_r are such that

$$V_r = V_v \cup V_e \qquad (2)$$

The vertices V_e, representing tasks and MediaPipes, are always leaves of T_r. The vertices V_v, representing resource partitions, are called *virtual resources*. An activity making use of a virtual resource is represented by a directed edge in T_r. There are some special vertices in V_v which are called *schedulers*. They are specialised virtual resources that can share their resource partitions among other virtual resources and schedulers. The sharing relationships are also represented by directed edges in T_r.

Analogous to components, providers, tasks and MediaPipes, composite resources can also be defined. This implies that a virtual resource in a resource tree T_r may represent a combination of different resource partitions. Indeed, composite virtual resources can also be shared when they are specialised as schedulers. ATM PVCs and MPLS traffic trunks are good realistic examples of schedulable composite virtual resources.

2.4. Meta-services

In SCM, adaptations of communication systems are modelled basically as changes to metadata. Such abstraction subsumes rather different adaptation mechanisms, ranging from those in charge of MediaPipe and task creation and deletion (e.g. signalling) to more radical ones such as changing a component (e.g. adding/ removing ports). In SCM, all kinds of adaptations are based on the concept of *open implementation* [35]. Besides the *base-level* (BL) ports that allow "normal" attachments and interactions, components may also have *meta-level* (ML) ports that reveal some of their internal aspects, thus allowing adaptations. Fig. 5(a) illustrates this concept. Components offering ML ports are called (adaptation) *targets*.

If a target component attaches an ML port to an access point of a provider, it will be offering another component, called *meta-component*, the opportunity to adapt its internals. Importantly, a meta-component can perform adaptations on components pertaining to any level of nesting.

As any component, a meta-component may communicate with other meta-components through a provider, thus defining a *meta-system*. Signalling systems and protocols such as SS7 [27] and RSVP [8] are good examples of meta-systems. The provider that the meta-components use to communicate may be deemed independent of the target system, as represented in Fig. 5(a), in which case it is called *meta-provider*. However, in some cases it is also useful to represent direct interactions among user/ service and meta-components through one single provider, as for example in reflective systems [38].

Meta-systems can also be targets of other meta-systems, constituting what is called a *meta-system tower*. The simpler representation of Fig. 5(b) is defined to represent a meta-target relationship between two systems without regard to specific components or providers.

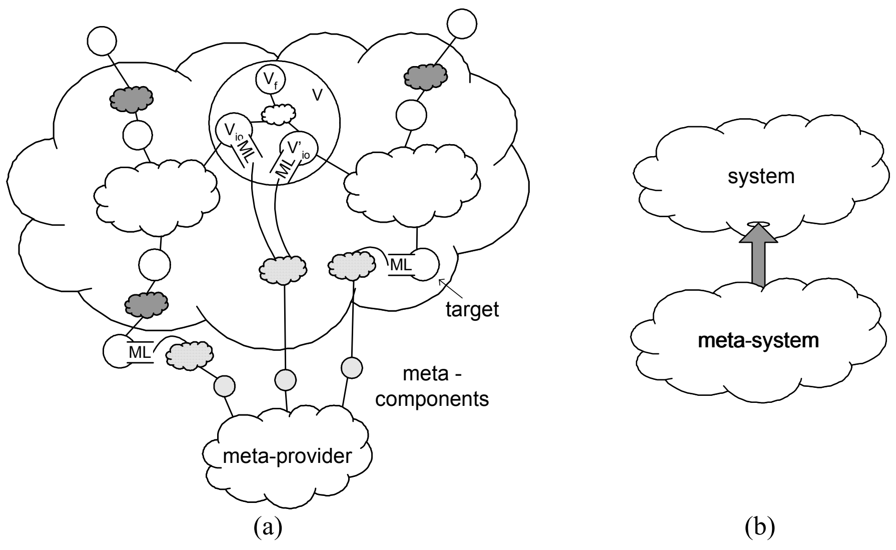


Fig. 5. Meta-systems.

To allow the explicit representation of meta-systems in the LindaX ADL, interface type declarations can have an attribute *Level* indicating whether the type is of meta-level. Going on with the example of the packet forwarding entity in Section 2.1, the components V_{io} and V'_{io} can both support a meta-level interface that is used to install packet filters, as illustrated in Fig. 5(a). Thus, the designer may declare a packet-classifying meta-level interface type, and make the type of components V_{io} and V'_{io} implement ports of this meta-level interface type, as illustrated in Fig. 6.

```
InterfaceType IMetaClassifier {
  Level = "meta"
  ...
}
ComponentType UIOModule {
  Port io {
    Cardinality = 2
    Signature in  { Type = PPacketPush }
    Signature out { Type = PPacketPull }
  }
  Port cls_ml {
    Cardinality = 1
    Signature cls { Type = IMetaClassifier }
  }
}
```

Fig. 6. Declaration of a meta-level interface type.

3. Adaptation planning with frameworks

Although SCM provides an approach to design communication systems and their adaptation mechanisms further structuring should be offered to service designers to regulate these adaptations. The generality of the model makes it infeasible to represent domain-specific constraints on adaptations. For example, without the help of further constraining abstractions, one cannot explicitly state in SCM that undesirable loops into a pipelined architecture such as that of a router's fast path are disallowed. Thus, providing the service designer with *support for expressing constraints* is essential if meaningful adaptation is to be represented. The notion of *frameworks* has therefore been applied in SCM to provide such support.

From the viewpoint of SCM, frameworks are semi-finished architectures, as defined by Pree [49]. They capture domain-specific design decisions that can be planned during design time, leaving 'incomplete' those parts of the system that are prone to adaptations – the so-called *hot spots*.

Hot spots permit the definition of adaptations throughout the whole life cycle (i.e. creation, deployment, operation, etc.) of a service. The phases of the life cycle in which the hot spots are 'filled' are also planned during design time, and will determine the level of adaptability of a certain system. For example, in previous work at PUC-Rio [21] a set of frameworks for QoS orchestration (see Section 6.1) have been defined with two different types of hot spots. The first type is in charge of treating specific service offering environment issues (e.g. whether using a sender- or receiver-oriented signalling protocol), thus providing for adaptations to be only performed during the creation of the service. The second type is responsible for regulating adaptability to new *categories*[5] of QoS requirements and allows adaptations performed during service operation.

The frameworks for QoS orchestration have been extensively used for earlier SCM-based implementations at PUC-Rio [36, 44, 46]. However, the lack of a *formal* approach to describe these frameworks has precluded their users (i.e. the designers) from effectively expressing constraints on adaptations. We argue that the use of formal framework descriptions permits the unambiguous interpretation of frameworks and the constraints they express. Moreover, a formal approach allows designers to reason about desired properties of the frameworks.

For the sake of formal description and reasoning of frameworks, LindaX focuses on the definition of *families* of architectures with sets of desirable properties that are common to a particular domain – the concept of *architectural styles*. A defining feature of LindaX is that the semantics of architectural configurations are totally based on the properties captured by their corresponding styles. This feature will be further discussed in Section 6.

LindaX supports the definition of styles with specific *vocabularies* and sets of *constraints*. The basic structure of a style declaration in LindaX is illustrated in Fig. 7.

[5] QoS categories are defined in [21] as sets of policies for QoS provisioning (e.g. resource scheduling and admission controlling algorithms) and the QoS parameters associated with these policies.

```
Style style-name {
  Superstyle = superstyle-name
  Vocabulary {
    ... /* type declarations */
  }

  Constraints {
    ... /* constraint declarations */
  }
}
```

Fig. 7. LindaX style declaration.

A vocabulary declares a set of component, provider and interface types known to the style. Constraints in LindaX impose restrictions on the ways the vocabulary of a style can be used. They represent rules that must be satisfied by any configuration that needs to conform to the style.

LindaX does not force the constraints into being specified with any particular notation; they can be populated with *predicates* describing the kind of notation used (e.g. free text, different classes of logic, etc). Analogous to types, predicates can be adorned with attributes.

The example of the packet forwarding entity is revisited to illustrate the applicability of LindaX styles. Fig. 8 depicts the skeleton of a style modelling a generic *forwarder*. Basically, a configuration conforming to the forwarder style must obey the following general rules:

- Compliant components must support appropriate numbers and combinations of ports of specific push- and pull-oriented packet-passing interface types.
- A compliant component may support a port of the packet-classifying meta-level interface type.

The style therefore declares a packet-classifying meta-level interface type and push-/ pull-oriented packet-passing base-level interface types as its vocabulary in addition to a single architectural constraint. This constraint is declared by a predicate of the type *FolPredicate*, which is defined in LindaX to allow the description of statements in first-order logic.

Styles can also be defined as extensions, or *sub-styles*, of another style. In the LindaX ADL, a sub-style makes reference to its super-style by means of an attribute *Superstyle*. A sub-style includes all the super-style's vocabulary and constraints. From the perspective of the abstract model, style hierarchies

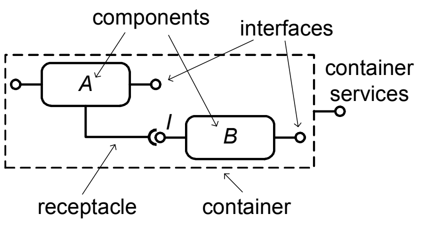
```
Style generic-forwarder {
  Vocabulary {
    InterfaceType PPacketPush {...}
    InterfaceType RPacketPush {...}
    InterfaceType PPacketPull {...}
    InterfaceType RPacketPull {...}
    InterfaceType IMetaClassifier
    {
      Level = "meta"
      ...
    }
  }

  Constraints {
    FolPredicate {
      forall c: Components {
        exists i: Ports(c) {
          ["PPacketPush" in Type(i)]
          or
          ["RPacketPush" in Type(i)]
          or
          ["PPacketPull" in Type(i)]
          or
          ["RPacketPull" in Type(i)]
        }
      }
    }
  }
}
```

Fig. 8. Skeleton of the forwarder style declaration.

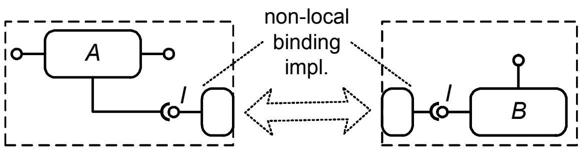
can be regarded as formal 'road maps' for filling particular hot spots. For example, a PC-based forwarder sub-style can be derived from the previous example, adding to it the following constraint:

- Any provider interconnecting packet-passing ports must offer minimal overhead (e.g. by disallowing undesirable context switches along the data path, or stating deterministic delay bounds for them).

It is important to notice the different levels of complexity between the above-mentioned constraint (which may involve reasoning about quantitative time) and the simple first-order logic predicate depicted in Fig. 8. LindaX can support different kinds of architectural reasoning by being extended with new predicate types as necessary.



(a)



(b)

Fig. 9. The concrete computational model.

4. The concrete model

Due to their highly abstract nature, the SCM concepts do not imply any particular computational model. This is usually a shortcoming when SCM is used as a basis for implementation[6]. In order to reduce such 'cognitive gap' between SCM-based specifications and low-level programming, this paper proposes the uniform use of a single component-based computational model, together with proper LindaX extensions and associated tools that help in mapping it onto SCM.

The proposal thus employs a concrete component-based computational model that is the basis of Lancaster's NETKIT toolkit [15]. This model offers a language- and platform-independent programming approach that can be applied at all levels of programmable networking from fine-grained, low-level, in-band packet processing functions, to high-level signalling and coordination functions. Moreover, the model offers, through open implementation and reflection facilities, flexible support for the deployment, instantiation and run-time adaptation of these functions.

The concrete model embodies the concepts of *software components*, *interfaces*, *receptacles*, *containers*, *local bindings* and *assemblies*. Fig. 9 illustrates these concepts. Unlike the abstract model, the concrete model defines components as purely software entities[7]. The concrete model also constrains the semantic-free concepts of ports and access points by defining interfaces and receptacles. Software components can support any number of interfaces and receptacles. Interfaces are immutable, strongly typed units of service provision. Each interface can support one of the following kinds of interaction: (i) traditional request/ reply operation invocations, (ii) streams, or (iii) signals. Receptacles are units of service requirement used to make explicit a dependency of one component on another. When a component is dynamically loaded it is possible to determine from its receptacles what other components/ interfaces must be present for the loaded component to work correctly. This is a crucial enabler for 'third-party' configuration and dynamic reconfiguration of component software, which is particularly interesting in the realm of programmable networking. For example, in Fig. 9(a) component *A* always needs the operations/ streams/ signals offered by interfaces of type *I*. It thus declares a receptacle of that type, so that when it is loaded its receptacle must be bound to an external interface instance of type *I* (in the example, provided by component *B*).

Containers delineate the concrete model view of a local service offering environment. The central role of containers is to provide generic services for dynamically loading and unloading software components, and for creating and destroying local bindings (see next paragraph). The container services are available from both inside and outside the container to support third-party loading and binding, for example, as part of a signalling procedure that loads a new scheduler component in a packet forwarder process. The concrete computational model is essentially *in-process*, that is, containers will be

[6] As already mentioned, several SCM-based prototypes have been implemented in different contexts at PUC-Rio. These prototypes range from support for packet and thread scheduling in operating systems [44], through signalling protocols for intserv and diffserv networks [46] to mobile management in mobile IP networks [36]. In all such pieces of work different approaches were used, including object-orientation (Java) and simple modular programming (ANSI C).

[7] Nevertheless, hardware entities can be wrapped as software components whenever necessary (e.g. for managing network cards).

typically (but not necessarily[8]) implemented as in-process address spaces.

Local bindings are associations between receptacles and interfaces that reside in the same container and are type compatible. They are assumed to be implemented minimally and with negligible or low overhead, to make it viable to be applied in demanding areas such as in-band packet processing. Non-local bindings (that is, across container boundaries, such as inter-process communication) and bindings with added semantics (e.g. multicast) are assumed to be built on top of the basic concrete computational model as software components. Fig. 9(b) illustrates the idea of non-local bindings. In fact, component-based distributed platforms (with support for non-local bindings) can also be built in terms of the model[9]. This approach differs radically from other middleware-based component models where components only appear on top of a 'black-box' distributed platform (e.g. the CORBA Component Model [47] and Sun's Enterprise JavaBeans [56]), since the middleware itself can be regarded, and consequently reconfigured, as a set of interconnected ('*plugged'*) components. This is similar in concept to the abstract notion of service providers; higher-level services are recursively built on lower-level services by using the same set of abstractions, which allows homogenizing the treatment of various aspects related to adaptation. Section 6.3 illustrates the implications of such feature with regard to QoS provision with a detailed example.

Finally, assemblies describe scopes of interaction among components serving some common purpose. An *encapsulated assembly* is referred to as one that can be represented by a single, composite component. In that case, the composite component offers interfaces and receptacles by exposing some of those of its constituents. Assemblies may cross container boundaries; non-local bindings are good examples of composite components whose constituents can be distributed along different containers.

4.1. Open implementation and reflection

The notions of open implementation and reflection are employed in the concrete model to allow both

[8] For example, a packet forwarder process can be distributed among a pipeline of network processors in the same router and still be managed by the same container services.

[9] This is one of the defining characteristics of Lancaster's OpenORBv2 component-based middleware platform [5].

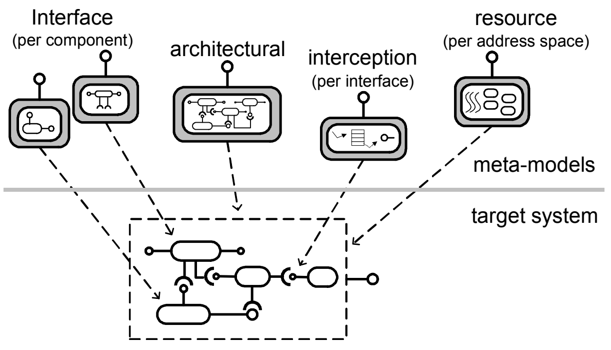


Fig. 10. Concept of meta-models in the concrete model.

inspection of current component configurations and subsequent reconfigurations. This is achieved by defining sets of *meta-components* and *meta-interfaces* that constitute *meta-models* of the system. Meta-models maintain metadata about the current configuration of components, monitor significant events and effect changes on (assemblies of) components. A defining feature of *reflective* meta-models is that they relate to the system in a causally connected manner, that is, a change made to metadata that a reflective meta-model manages implicitly causes a corresponding change in the underlying system, and vice versa.

Generic support for reflection is provided in the concrete model by four *core* meta-models [5], as depicted in Fig. 10: *interface*, *architecture*, *interception* and *resource* meta-models.

The interface meta-model supports inspection and limited adaptation (i.e. showing/ hiding) of interface and receptacle information on a per-component basis.

The architectural meta-model is used to reify the abstract topology structure of a system on a per-component (i.e. assemblies) or per-container basis. It serves as a central point for architectural reconfiguration, managing the insertion/ removal of components and bindings in/ from configurations.

The interception meta-model enables the association/ dissociation of *interceptors* with/ from operations/ streams/ signals of some particular interface. Interceptors are chunks of code that can be invoked before and/ or after every invocation on the specified interface. The interception meta-model is particularly interesting for the aims of this paper at constraints enforcement, as will be seen in Section 5.

Finally, the resource meta-model is used to reify the abstract resource structure of a system. This meta-model provides a flexible and fine-grained resource management scheme by realising the abstract concept of tasks (see Section 2.2) into concrete, dynamic entities. More details about the resource meta-model can be found in [19].

4.2. Implementation

The concrete model has been adopted in a platform implementation, called OpenCOM, which was developed from previous research in configurable middleware at Lancaster [13]. OpenCOM was initially built atop a subset of Microsoft's COM [41]. Work is currently being progressed to adapt OpenCOM to better support programmable networking software, by freeing it from COM dependencies and porting it to other platforms[10]. At present, OpenCOM comprises a small number of low-level 'core' aspects: (i) COM-based binary-level interoperability standard (i.e. the *vtable* data structure) and globally unique identifier schema (i.e. the GUIDs), (ii) a CORBA-compliant IDL, (iii) a set of platform-independent container services and (iv) the *IReference* base interface (for reference counting and distributed garbage collection), from which all other OpenCOM interfaces must derive.

OpenCOM deploys a standard run-time in every container. This is implemented as a primitive, platform-specific component that provides an interface for the container services. The primary role of the run-time is to manage a repository of available component types and thus support the creation and deletion of component instances. The run-time offers the minimal functionality of OpenCOM, upon which relies the implementation of the core reflective meta-models, as will be seen in the following section.

5. Enforcing adaptation constraints: frameworks revisited

Parlavantzas et al [48] state that, although *necessary*, the explicit representation of dependencies and the reflective meta-models of the concrete model are not in themselves *sufficient* for imposing domain-specific constraints and policing adaptations. For example, they cannot mandate that a packet scheduler must always receive its input from a packet classifier within a router. Such type of constraints is essential to avoid nonsensical adaptations.

To add the necessary *support for enforcing constraints* a component-based notion of frameworks is applied in the concrete model. As defined by Szyperski [57], *component frameworks* (CFs) govern the interaction of a set of 'plugged' components (*plug-ins*) by means of a collection of domain-specific rules and interface types. Putting it another way, the design of CFs in the context of the concrete model allows the development of bespoke meta-models that make sense in a particular domain. So for example, in a related project at Lancaster [16] a protocol meta-model is employed that embodies knowledge, in the form of appropriate rules and interfaces, about the configuration and reconfiguration of a set of plug-ins representing protocol entities in a layered architecture.

A defining characteristic of CFs is that a CF instance (i.e. an assembly that conforms to the rules of the CF) is explicitly represented at run-time by a component – a *CF representative* (CFR) – that maintains metadata about the CF instance and its rules. In fact, OpenCOM itself can be seen as a 'zero-level' CF (the run-time component being the CFR) that is in charge of managing type libraries and loading components within a single container. Likewise, the core reflective meta-models can be seen as 'first-level' CFs with their respective meta-interfaces and CFRs being responsible for providing generic support for reflective adaptability. Above that, a set of domain-specific CFs can be defined to leverage pieces of higher-level functionality.

As a general rule, higher-level CFRs offer domain-specific functionality (and often domain-specific meta-interfaces as well) built on lower-level meta-models. For example, considering again the protocol meta-model mentioned above, the associated CFR could build on the causally connected topology structure of the reflective architectural meta-model to offer an implementation of the architectural meta-interface specialised in manipulating the protocol stack. This could be accomplished by means of interceptors that redirect the operation invocations on this meta-interface to the CFR. The CFR could then exploit its implicit domain-specific knowledge to manage the requested reconfiguration operations with minimum perceived disruption of the data flows that go across the stack; by buffering data in the meanwhile, for example.

[10] Currently, there is an implementation of OpenCOM running both on Windows and Linux. See [33] for a discussion about the ongoing implementation of OpenCOM on Intel's IXA network processor architecture [25].

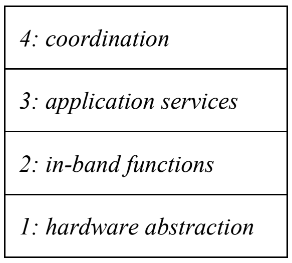


Fig. 11. Software stratification of programmable networking. The term 'stratum' is used rather than 'layer' to avoid confusion with layered protocol architectures.

That general rule is being used in the NETKIT project to organise the design of domain-specific CFs according to the broad-brush stratification illustrated in Fig. 11. The hardware abstraction stratum (stratum 1) comprises the minimal operating system (OS) functionality (e.g. threads, memory allocation, and access to network hardware) that must be available on any participating node (e.g. router) to support higher-level network programmability. Second, the in-band functions stratum comprises packet processing functions (e.g. packet filters, checksum validators, classifiers, diffserv schedulers, shapers, etc.) that touch all packets. Third, the application services stratum comprises coarser-grained 'programs' — in the active networking execution-environment sense [60] — that are less performance critical and act on pre-selected packet flows in application-specific ways (e.g. per-flow media filters). Finally, the coordination stratum comprises out-of-band signalling protocols that perform distributed coordination and (re)configuration of the lower strata, such as RSVP or protocols that coordinate resource allocation in dynamic private virtual networks [11].

We envision LindaX subsuming in a single notation the views of frameworks as *both* abstract semi-finished architectures *and* collections of domain-specific types and constraints for the concrete model. Therefore, besides being possible to use LindaX for formally specifying and reasoning about frameworks, LindaX style descriptions can also be translated into concrete CFs. More specifically, LindaX supports the generation of NETKIT-based stratum-specific meta-models and their related CFRs from style definitions. To that end, LindaX offers a set of translation conventions and structures.

First, a mapping scheme between entities in the abstract and concrete models has been defined. The mapping of abstract components and ports onto, respectively, concrete software components and interfaces/ receptacles are relatively straightforward. Providers are mapped onto bindings by using the following rules:

- The absence of provider type declarations in a style implies the use of bindings without added semantics. These bindings can be local or not; in the latter case, the non-local bindings are implemented in the concrete model as software components. This requires human intervention in the generation process for choosing the most appropriate binding component implementation.
- When a style vocabulary includes provider types, the corresponding binding component implementation is identified by using LindaX extensions, as defined below.

Second, LindaX extensions have been introduced to make correspondences between the LindaX and OpenCOM type systems. One such extension is the attribute *Id* in the vocabulary declaration (see Fig. 13), which makes an explicit reference between OpenCOM component/ interface type GUIDs and the style vocabulary. For example, this permits that during the insertion of a new OpenCOM component in a router CF instance the associated CFR uses the interface meta-model of the candidate component to assess whether it implements the required packet-passing interfaces. The attribute *Id* is also important during architectural reconfigurations involving the creation of 'richer' bindings: it can help a CFR in identifying the correct binding component implementation to be used when the associated style vocabulary includes a specific provider type.

Third, the concept of *constraint scopes* has been defined. A constraint scope relates a particular constraint to interface types that are marked as of meta-level. To date, there are two basic constraint scopes:

- A constraint is associated with *all* meta-interface types (the default case).
- A constraint is associated with a particular meta-interface type.

Such concept is used to generate domain-specific CFRs; in the adopted approach all operations/ streams/ signals within a particular constraint scope can be instrumented with invariant checks. Each invariant is in their turn produced from the predicate associated with the scope (by the use of a transformational approach [59], for example). For the sake of experimenting with the use of constraint scopes, LindaX has been extended by the introduction of the attribute *Scope* into the predicate type *FolPredicate*. Our approach to constraints enforcement in CFs is then built on two mechanisms: (i) the transactional reconfiguration process proposed by FORMAware, a CF for safely configuration management developed at Lancaster [43], which permits the initiation, commitment and rollback of reconfiguration transactions in OpenCOM; and (ii) when the scoped operations/ interfaces make part of one of the core reflective meta-models, the interception meta-model is used to redirect the invocations to the CFR.

As an example of the applicability of LindaX in the NETKIT project, the forwarder style defined in Section 3 can be refined so as to model the stratum-2 router CF presented in [15]. An instance of that CF accepts, as plug-ins, components that perform arbitrary packet-forwarding functions. The following CF rules are added to the general constraints of the forwarder style:

- A router CF instance must be always encapsulated (i.e. either a simple or composite component);
- When the CF instance is a composite, all its internal constituents must (recursively) conform to the CF rules. In such cases, the functionality of the packet-classifying meta-level interface is implemented by the associated CFR, which processes requests coming from that meta-level interface and forwards them to composite's constituents as appropriate. The CFR must also intercept operation invocations on the meta-interface of the reflective architectural meta-model in a way that it only allows conforming reconfigurations of the CF topology (see Fig. 12 for an example of a conforming composite).

Fig. 13 exemplifies the use of LindaX for specifying the router CF. In the figure it is stated, by means of the attribute *Scope*, that the first-order logic predicate previously depicted in Fig. 8 must be enforced for all operations of the architectural meta-interface so that they can be instrumented with invariant checks synthesised from the predicate.

6. Bespoke architectural description support

Contrasting with traditional ADLs, which are usually based on general purpose artefacts (i.e. *components* and *connectors*), LindaX makes use of

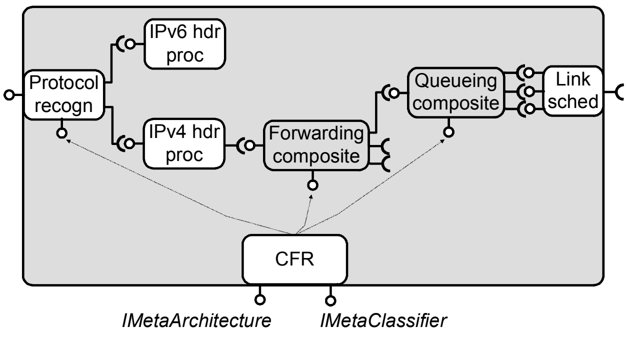


Fig. 12. A composite component conforming to the router CF.

the vocabulary and set of constraints of a particular style to offer a level of specification for architectural configurations which is closer to the domain targeted by the style. The rationale behind this resides in the fact that, in many situations, the inherent dynamism and complexity of programmable networks may hamper an effective use of ADLs providing the generic abstraction 'components linked by connectors'. In previous work at PUC-Rio, Soares-Neto [54] describes the frameworks for QoS orchestration in the Wright ADL [2], showing that the application of traditional ADL artefacts in the realm of adaptable communication systems may result in too complex, lengthy, and consequently error-prone specifications. Soares-Neto then proposes a domain-specific language – called *LindaQoS* – to simplify those specifications based on the frameworks for QoS orchestration. A compiler is implemented to translate LindaQoS specifications into Wright descriptions.

Building on Soares-Neto's work, our proposal relies on LindaX offering a *style-specific configuration support* that (we believe) leads to simple and concise architectural descriptions. However, LindaX provides a more generic syntax that can embrace architectural descriptions in a broader set of domains. Thus, the LindaX configuration support acts like a template for the definition of descriptions with domain-specific semantics. The semantics of an architectural description in LindaX rely on interpreters/ compilers that are specialised in specific styles, as will be exemplified in the remainder of this section.

Configurations in LindaX can include *components*, *systems, subsystems* and *links*. The basic structure of these configurations is show in Fig. 14.

```
Vocabulary {
  InterfaceType PPacketPush {
    Id = 1724c31e-48dc-45ee-858e-acd7f3618383
    ...
  }
  InterfaceType PPacketPull {
    Id = ea78e8b2-1552-4a76-8cf7-9bc13626bc33
    ...
  }
  InterfaceType IMetaArchitecture {
    Level = "meta"
    Id = 09209844-a6b2-454e-b0c2-f720936af555
    ...
  }
  ... /* other type definitions */
}
Constraints {
  FolPredicate {
    Scope = "IMetaArchitecture"
    forall c: Components {
      exists i: Ports(c) {
        ["PPacketPush" in Type(i)]
        or
        ["RPacketPush" in Type(i)]
        or
        ["PPacketPull" in Type(i)]
        or
        ["RPacketPull" in Type(i)]
      }
    }
  }
  ... /* other constraint definitions */
}
```

Fig. 13. Use of attributes *Id* and *Scope*.

A component represents a unit of computation whose semantics are defined by a *reference* to a component type (declared as part of the vocabulary of the conformer style). A system is a configuration of components that conforms to some style. A subsystem is a system that can be in turn treated as a single entity (e.g. a composite component) by an enclosing system. A link is a direct association between components and subsystems, thus masking the existence of explicit communication entities (i.e. service providers and MediaPipes). Virtually all these structures can be parameterised. Again, the language gives no general support for assessing links (e.g. whether there is a link between incompatible interface types) or validating parameters, so that any check is left to style-specific tools.

```
System sys1-name { ... }

System sys2-name {
  Style = style-name
  Component comp-name { Type = type-name }
  Subsystem subsys-name { Type = sys1-name }
  Link { comp-name, comp-name }
  Link { comp-name, subsys-name }
  Link { subsys-name, subsys-name }
}
```

Fig. 14. LindaX configuration support.

6.1. LindaQoS v2.0

As an example of our approach, we have re-engineered LindaQoS by developing LindaX styles and specialised LindaX configuration description compilers – the so-called LindaQoS v2.0 – for the frameworks for QoS orchestration. These frameworks build on the recurrent nature of SCM by providing a basic semi-finished architecture that can be adapted and recurrently applied to model complex QoS orchestration scenarios involving many different subsystems.

The process of QoS orchestration is a typical example of meta-system, acting upon a (target) system in two main phases: *QoS negotiation* and *tuning*. The QoS negotiation phase involves mechanisms responsible for the admission of new tasks and MediaPipes (with specific QoS requirements) to a running system. Resource reservation and commitment are main goals in this phase. To that end, there is an admission process that establishes a service agreement on the use of resources; tasks and MediaPipes must not make use of more resource capacity than they are supposed to (according to their QoS requirements), otherwise they are subject to a disruption to their service agreement. The QoS tuning phase provides mechanisms responsible for monitoring the use of resources after the establishment of a service agreement, and in case of violation of the agreement certain actions may be triggered, ranging from simple notifications to overall re-orchestration. To date, we have focused on the styles and compilers related to the QoS negotiation mechanisms only. Information on the specification of

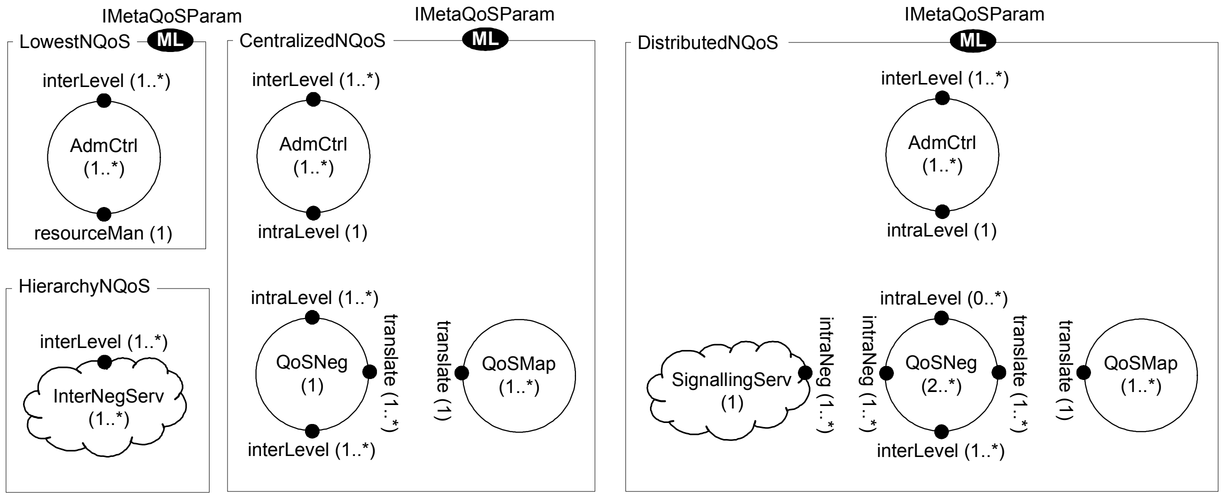


Fig. 15. Informal graphical notation for LindaQoS negotiation styles.

QoS tuning mechanisms in LindaQoS v1.0 can be found in [54].

Three main meta-component types define the negotiation architecture in the QoS orchestration meta-system: *admission controllers*, *QoS negotiators* and *QoS mappers*. A request for a service agreement is done through a call to an admission controller. If this admission controller is related to a composite resource (see Section 2.3.2), it starts an associated QoS negotiator. The negotiator identifies[11] the internal resources that can be involved in the provision of the required service and divides the portions of QoS responsibility among them. A new request for service agreement is then done on each internal resource, through their corresponding admission controllers. Since the QoS requirements are usually represented in a different way for each type of resource involved, QoS mappers are used to translate the QoS requirements accordingly. The admission controllers related to each internal resource repeat the admission process recurrently, eventually reaching admission controllers that are directly associated with primitive resources. These admission controllers will typically act upon metadata in the resource structure to provide for resource reservation and commitment.

[11] Other mechanisms can be involved in such identification, such as routing. This is out of the scope of this paper though.

6.2. LindaX styles for QoS negotiation

Four styles have been defined for the specification of the negotiation architecture of the QoS orchestration meta-system: *LowestNQoS*, *CentralizedNQoS*, *DistributedNQoS* and *HierarchyNQoS*. For conciseness, a full description of these styles in LindaX is omitted. The styles are described by using an informal graphical notation, as depicted in Fig. 15. The conventions adopted in the figure are as follows.

First, circles represent component types. The component type name and the component cardinality are indicated inside the circle, the cardinality within parenthesis. Dots around a circle represent port signatures, and the port name and cardinality are indicated next to the dot. Similarly, clouds represent provider types, and dots around a cloud represent access point signatures. Importantly, no type in these styles is previously bound to any specific implementation (i.e. the attribute *Id* is parameterised), so that the same style can be used for generating code for different parts of the QoS negotiation meta-system.

Second, a 'nomenclature' for port names has been adopted in which the *intra* prefix indicates ports and providers used for communications among meta-components acting upon the same target subsystem, whereas the *inter* prefix indicates communications among meta-components acting upon different target subsystems. Finally, the first three styles declare a meta-interface type *IMetaQoSParam* (represented by

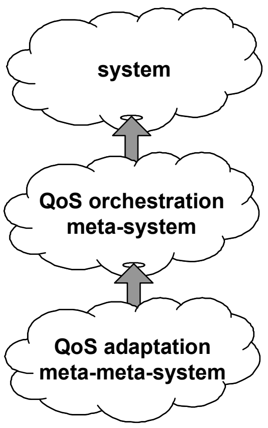

system
QoS orchestration meta-system
QoS adaptation meta-meta-system

Fig. 16. QoS meta-system tower.

a dot with an adornment "ML"). CFRs generated from these styles will use interfaces of this type as a central point for 'filling' the hot spots in charge of providing adaptability to new categories of QoS requirements. It is important to notice that the generated CFRs will constitute meta-models of the QoS orchestration meta-system, thus defining a meta-system tower, as depicted in Fig. 16.

The *LowestNQoS* style describes parts of the meta-system that make the admission control and resource allocation directly over primitive resources. In such parts admission controllers are the only meta-components. They are responsible for linking the negotiation architecture to the target system, what is represented by the port of type *ResourceMan*. Ports of this type must always be connected with an ML port providing resource management services: for example, such ports can be mapped onto OpenCOM receptacles for meta-interfaces of the resource meta-model (*IResourceManager*).

The *CentralizedNQoS* style describes parts of the meta-system that centralise the negotiation procedure in one single meta-component. The COPS framework for policy-based admission control [20] is a good example of where this style could be applied. The central QoS negotiator meta-component can receive requests for new service agreements from one or more admission controllers (through ports of type *IntraLevel*), and forward these requests to admission controllers in other parts of the meta-system (through ports of type *InterLevel*). For each of those 'external' admission controllers, there is an associated QoS mapper meta-component.

The *DistributedNQoS* style represents those parts of the meta-system in which the negotiation procedure is distributed among several QoS negotiator meta-components, such as in RSVP-based networks. One of these meta-components (typically, but not necessarily, one residing in an end-system) can receive requests from admission controllers, distribute these requests among other QoS negotiator meta-components (through meta-providers of type *SignallingServ*) and forward these requests to other parts of the meta-system, in a similar way as the centralised case.

Finally, the *HierarchyNQoS* style provides a kind of 'glue' among all parts of a meta-system. This style defines the provider type *InterNegServ*, which regulates the interaction among QoS negotiators and admission controllers in distinct parts of the meta-system. This particular way of representing 'glues' as a separate style results from the lack of recursive style support in LindaX.

6.3. Example: a programmable RSVP-based signalling system

Fig. 17 gives an example of an architectural configuration description in LindaQoS v2.0. The described meta-system encompasses the QoS negotiation mechanisms acting on two end-systems and a router connecting them. The overall QoS negotiation is done in a distributed way by an RSVP-based meta-system (*RSVPNetwork*), which coordinates the resource management mechanisms in the two end-systems (*EndSys1* and *EndSys2*) and the router (*Router*). Each of the three systems has an OS QoS negotiation subsystem (*OS**), which in their turn manages two kinds of resources: CPU time (*CPUManager**) and link bandwidth (*LinkManager**).

As Fig. 17 illustrates, the *AdmCtrl* and *QoSNeg* type references are parameterised. In LindaQoS v2.0, the only parameter passed to *AdmCtrl* type references is the GUID of the OpenCOM component that implements the admission controller in that system. *QoSNeg* type references accept two parameters: the GUID of the component implementing the negotiator, and a list of GUIDs of the components implementing mappers for different categories of QoS requirements. The *RSVPNetwork* system declaration is also parameterised; a single parameter is accepted for distributed styles, which indicates the GUID of the binding component implementing the signalling protocol (in the example, RSVP).

```
System CPUManager1 {
  Style = LowestNQoS
  Component ca { Type = AdmCtrl(...) }
}
 System LinkManager1 {
  Style = LowestNQoS
  Component ca { Type = AdmCtrl(...) }
}
System OS1 {
  Style = CentralizedNQoS
  Component ca  { Type = AdmCtrl(...) }
  Component neg { Type = QoSNeg(...) }
  Link { ca, neg }
}
System EndSys1 {
  Style = HierarchyNQoS
  Subsystem cpu1 { Type = CPUManager1 }
  Subsystem lnk1 { Type = LinkManager1 }
  Subsystem os1  { Type = OS1 }
  Link { os1.neg, cpu1.ca }
  Link { os1.neg, lnk1.ca }
}
...//same for the other end-system (EndSys2)
   //and the router. Router has 2 links
System RSVPNetwork(...) {
  Style = DistributedNQoS
  Component ca1  { Type = AdmCtrl(...) }
  Component ca2  { Type = AdmCtrl(...) }
  Component neg1 { Type = QoSNeg(...) }
  Component neg2 { Type = QoSNeg(...) }
  Component negN { Type = QoSNeg(...) }
  Link { ca1, neg1 }
  Link { ca2, neg2 }
  Link { neg1, negN }
  Link { negn, neg2 }
}

System All {
  Style = HierarchyNQoS
  Subsystem es1 { Type = EndSys1 }
  Subsystem es2 { Type = EndSys2 }
  Subsystem rt  { Type = Router }
  Subsystem rsvp { Type = RSVPNetwork }
  Link { rsvp.neg1, es1.ca }
  Link { rsvp.neg2, es2.ca }
  Link { rsvp.negN, rt.ca }
}
```

Fig. 17. Example of architectural description in LindaQoS.

7. Related work

7.1. Programmable networking

Historically, there have been two main paradigmatic approaches to the provision of openness and programmability in networks: *active networking* and *open signalling*. In a nutshell:

- Active networking (e.g. the ANTS toolkit [60] and the Smart Packets project [53]) is more dynamic but it is perceived as more prone to security threats and is more likely to be language-specific.
- Open signalling (e.g. the Genesis kernel [11] and the VServ architecture [31]) is easier to secure and typically performs better, especially at the level of critical elements like classifiers, but it is less dynamic.

Recently, new proposals have been made to try to converge the two paradigms: the main idea is that open signalling routers would also support downloadable modules and therefore would be more dynamic (e.g. the NetBind component binding system [12]). This leads to a third approach, so called *out-of-band active*, in which programmable routers differ in their support for kernel vs. user space modules, and in the way in-band functions can be adapted.

It is interesting to notice that such approaches tend to address only a subset of the concerns implied in the stratification proposed at Section 5. More specifically, active networking research tends to focus on OS support for active application services (i.e. strata 1 and 3), whereas open signalling research typically focuses on in-band packet-processing and out-band coordination functions (i.e. strata 2 and 4, respectively). Out-of-band active proposals, despite being regarded as a combination of the two approaches, typically leave strata 1 and 4 functionality to, respectively, legacy OSs and the ‘application’.

Other pieces of work that are acknowledged as paradigm-independent have limited coverage in at least one of the following aspects:

- Platform diversity: for instance, the Click modular router [45] focuses on PC-based routers.
- Level of programming abstraction: the NetBind, the VERA extensible router architecture [34] and the LARA++ architecture [52], for example, have more comprehensive models, but provide much

lower-level and less general abstractions for network software composition.

- Reconfiguration management: none of the above-mentioned work explicitly supports the management of system integrity over reconfigurations; in other proposals (e.g. the Pronto platform [23]) some support for integrity checking is given, although again with limited scope (typically, focusing only on active application service and coordination function adaptations, that is, strata 2 and 4 functionality).

Overall, what appears to be missing from the state of the art is an integrated solution that: (i) offers a paradigm-, platform- and language-independent programming model, (ii) can be uniformly applied throughout the programmable networking design space, and (iii) explicitly supports reconfiguration management. We argue that the proposed NETKIT toolkit provides a promising approach to achieve such solution.

7.2. Service creation

Despite the above-mentioned efforts to provide good support for implementing adaptable communication systems software and managing its evolution, the need for formal verification and validation of such sort of software has been recurrently advocated in the literature as mandatory if network operators are loath to damage their market shares. Much of the work in this area fits into the following classification:

- *Formal description techniques* (FDTs – e.g. LOTOS [7], Estelle [9], SDL [28], CSP [24]) allows the representation of the behaviour of a system in an abstract level, independent from its implementation, thus providing a basis for the analysis of the system prior to its development.
- *Architecture description languages* (ADLs - e.g. Darwin [39], Wright [2], Rapide [37]) provide declarative notations for decomposing a system into components and connectors and specifying how these elements are combined to expose the system software architecture. Moreover, some ADLs (e.g. Wright) allow the definition of families of software architectures through architectural styles. These characteristics offer a means of evaluating the design (and even validating it, when ADLs are used together with FDTs), guiding detailed implementation of the system, and improving design and implementation reuse.

FDTs are in general quite adequate for representing adaptable communication systems due to their ability to continuously verify and validate system properties. However, they are so generic that the specification of a system turns out to be excessively complex and costly (a detailed survey on FDTs for communications services is found in [18]). Conversely, ADLs are easier to use, but although many ADLs also give support to dynamic architectures (e.g. Darwin), such support has a rather more limited scope than FDTs (a thorough comparison among several ADLs can be found in [40]). Aujla et al [3] express the need for a gradual integration of FDTs into system development. The decision on a modular development of LindaX concerned such statement, since formalisms can be introduced as needed. LindaX has been built as a set of extensions to xArch [30], an XML-based representation for software architectures that can serve as a starting point for more advanced XML-based architectural notations. Moreover, we have adopted the same modularity philosophy as xADL [17], an extensible ADL developed from xArch, by deploying an extensible LindaX core. Specifications based on xArch are supported by generic tools (the ArchStudio v3 tool suite [29]) that provide storing, manipulating and sharing facilities. These have been extensively used for LindaX.

Besides the general considerations taken above, other specific architectural features required by adaptable communication systems do deserve further attention. These are investigated in detail in [22], so that they are only commented in brief below.

First, a few ADLs such as Aster [32], XelHa [19] and Olan [4] take into account non-functional properties that are defining characteristics of communication systems, such as resource management and QoS specification. Even so, the artefacts provided are rather limited; for example, XelHa offers a flat QoS tuning architecture that is not enough to represent tuning scenarios involving various subsystems. In this way, LindaQoS is built on a far more general model for QoS negotiation and tuning. Moreover, in spite of these ADLs being able to represent adaptations, they do not explicitly offer specific adaptation interfaces, as does LindaX, so that there is no easy way for the designer to identify points of adaptation.

Second, the need for *both* correctness verification *and* rapid deployment of telecommunications services implies formalisms and high-level architectural descriptions that can be simultaneously targeted at both needs. However, most of the existing work privileges one against the other (e.g. Wright, Darwin and Rapide focus on formal analysis, whereas Aster, Olan and XelHa are rather aimed at systematic synthesis of systems). The work by Moreira et al [43] goes in a different direction by proposing the explicit representation of architectural styles in a computational model. This solution provides the system with awareness about its own architecture and a principled way to deal with adaptation, which is rather similar in concept to our notion of CFRs being derived from styles. However, the styles are described as chunks of program (Java classes), what is quite a limitation for the purposes of correctness verification. Again, since LindaX builds on xArch extensibility we believe it is not much complicated, to some extent, to combine formalism and system synthesis in LindaX.

The Knit [50] and GenVoca [62] systems are both aimed at generation of component-based software systems. Knit promotes reuse of existing code (in C) by means of composition rules based on a linking language. In the GenVoca approach, system code is synthesised from a high-level (and domain-specific) program description. Our approach for system synthesis is based on tools, plugged into the ArchStudio tool suite, which take a coarser-grained method: rather than synthesising *all* code out of LindaX descriptions, we aim at using libraries of components to be provided by NETKIT and use our tools to synthesise code (in the form of CFRs) for *composing* them.

8. Final remarks and conclusions

From previous experiences in building both SCM- and OpenCOM-based software systems, we believe that both models, in themselves, do not completely address the needs of software architecture for programmable networks. The SCM concepts provide a good abstract model for formally specifying architectures, but due to their generality SCM lacks a well-defined computational model that helps designers in implementation. On the other hand, the notion of CFs in OpenCOM provides a solid basis for the implementation of constraints enforcement; nevertheless, no formal basis is provided in OpenCOM for architectural reasoning of CFs. Aiming at filling this gap, we have developed LindaX, an ADL targeted at the formal specification of frameworks as architectural styles, and of architectural configurations as domain-specific structures whose semantics are dependent on a specific style.

Although the main target presented in this paper is the creation and deployment of OpenCOM-based software for programmable networks, the LindaX ADL is highly extensible, so that it can target diverse service offering environments. In special, current work is being done at PUC-Rio on implementation of LindaX and LindaQoS compilers that can reuse most of the existing Java and C code of SCM-based prototypes.

In an analogous way, the LindaX ADL is aimed at encompassing different formalisms. To date, the only support for architectural reasoning in LindaX is offered by the first-order predicate type *FolPredicate*, which does not present to the designer a sufficiently powerful set of possible constraints. We have been assessing the use of the real-time temporal logic QTL [6] to specify real-time constraints. These are particularly interesting for the description of *performance objectives*: for example, in the styles for centralised and distributed QoS negotiation a real-time constraint could be declared to impose a deterministic maximum delay on the establishment of service agreements. The idea of using QTL is closely related to our interest in execution configuration descriptions, as described further bellow.

In addition to formal descriptions of style constraints, we have progressed work on describing the behaviour of components, providers and interface types in LindaX as events in a CSP-based notation similar to that used in the Wright ADL. The rationale behind this approach is that with a formal behaviour specification we will be able to reason not only about architectural properties (in a topological sense) of styles and configurations, but also to provide our tools with some additional analytic leverage (e.g. to prevent deadlocks).

We also think of extending LindaX to support the description of *execution* configurations besides *architectural* configurations. The idea of execution configurations resembles that of the task model in XelHa [19]: namely, the execution and derivation graphs of the abstract model (see Section 2.3.1) could be specified in a declarative way, much in the same manner that the architectural graph can be currently specified with LindaX architectural configuration support. Execution configuration descriptions would then allow service designers to more easily engineer the scopes for resource management in an adaptable communication system.

Other marginal interests of our work on LindaX in the future are: (i) the generation of LindaX style-specific configuration compilers, such as those for LindaQoS, (semi) automatically, and (ii) the definition of LindaX styles that permit the modelling of a QoS adaptation 'meta-meta-system' which manages, in an *integrated* way, the overall adaptation of QoS orchestration meta-systems to new categories of QoS requirements.

Acknowledgments

The authors would like to thank Carlos de Salles Soares-Neto for sharing his work on the Wright styles for QoS orchestration and the LindaQoS v1.0 compiler and also for providing all the necessary information about them.

Antônio Tadeu A. Gomes is sponsored by CAPES, Brazil.

References

[1] Agere Systems, Agere network processors. Available from <http://www.agere.com>.

[2] R.J. Allen, A formal approach to software architecture, Ph.D. Thesis, Technical report CMU-CS-97-144, School of Computer Science, Carnegie Mellon University, Pittsburgh PA, USA, May 1997.

[3] S. Aujla, T. Bruyant, L. Semmens, Applying formal methods within structured development, IEEE Journal on Selected Areas in Communications 12 (2) (1994) 258-264.

[4] L. Bellissard, N. de Palma, D. Féliot, The Olan architecture definition language, Technical report 24, ESPRIT Long Term Research Project 24962, 1998.

[5] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, K. Saikoski, The design and implementation of Open ORB version 2, IEEE Distributed Systems Online Journal 2 (6) (2001).

[6] G.S. Blair, L. Blair, H. Bowman, A. Chetwynd, Formal specification of distributed multimedia systems, UCL Press, London, UK, 1998.

[7] T. Bolognesi, E. Brinksma, Introduction to the ISO specification language LOTOS, in: P. Eijk, C. Vissers, M. Diaz (Eds.), The Formal Description Technique LOTOS, Elsevier, Amsterdam, 1989.

[8] R. Braden, L. Zhang, S. Berson, S. Herzog, S. Jamin, Resource ReSerVation Protocol (RSVP) – version 1 functional specification, RFC 2205, Internet Engineering Task Force, 1997.

[9] S. Budkowski, P. Dembinski, An introduction to Estelle: A specification language for distributed systems, Computer Networks and ISDN Systems 14 (1987) 3-23.

[10] A.T. Campbell, H.G. De Meer, M.E. Kounavis, K. Miki, J.B. Vicente, D.A. Villela, A survey of programmable networks, ACM SIGCOMM Computer Communications Review 29 (2) (1999) 7-23.

[11] A.T. Campbell, M.E. Kounavis, D.A. Villela, J.B. Vicente, H.G. de Meer, K. Miki, K.S. Kalaichelvan, Spawing networks, IEEE Network Magazine 13 (4) (1999) 16-29.

[12] A.T. Campbell, S. Chou, M.E. Kounavis, V.D. Stachtos, J.B. Vicente, NetBind: A binding tool for constructing data paths in network processor-based routers, in: Proceedings of the 5th IEEE Conference on Open Architectures and Network Programming (OPENARCH'02), New York, USA, June 2002.

[13] M. Clarke, G.S. Blair, G. Coulson, N. Parlavantzas, An efficient component model for the construction of adaptive middleware, in: Proceedings of the IFIP/ ACM International Middleware Conference (Middleware'01), Heidelberg, Germany, Lecture Notes in Computer Science 2218 (2001) 160-178.

[14] S. Colcher, A.T.A. Gomes, L.F.G. Soares, Um meta modelo para a engenharia de serviços de telecomunicações (english version: A meta-model for the telecommunication services engineering), in: Proceedings of the 18th Brazilian Symposium on Computer Networks (SBRC'00), Belo Horizonte, Brazil, May 2000.

[15] G. Coulson, G.S. Blair, A.T.A. Gomes, A. Joolia, K. Lee, J. Ueyama, Y. Ye, A reflective middleware-based approach to programmable networking, in: Proceedings of the IFIP/ ACM/ USENIX International Middleware Conference (Middleware'03) Workshops, Rio de Janeiro, Brazil, PUC-Rio, June 2003, pp 115-119.

[16] G. Coulson, G.S. Blair, M. Clarke, N. Parlavantzas, The design of a highly configurable and reconfigurable middleware platform, ACM Distributed Computing Journal, 15 (2) (2002) 109-126.

[17] E.M. Dashofy, A. van der Hoek, R.N. Taylor, An infrastructure for the rapid development of XML-based architecture description languages, in: Proceedings of the 22nd International Conference on Software Engineering (ICSE'02), Orlando FL, USA, May 2002, pp 266-276.

[18] F. Dietrich, J.-P. Hubaux, Formal methods for communication services: Meeting the industry expectations, Computer Networks 38 (1) (2002) 99-120.

[19] H.A. Duran-Limon, G.S. Blair, Reconfiguration of resources in middleware, in: Proceedings of the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'02), San Diego CA, USA, January 2002.

[20] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, A. Sastry, The COPS (Common Open Policy Service) protocol, RFC 2748, Internet Engineering Task Force, 2000.

[21] A.T.A. Gomes, S. Colcher, L.F.G. Soares, Modelling QoS provision on adaptable communication environments, in: Proceedings of the IEEE International Conference on Communications (ICC'01), Helsinki, Finland, June 2001.

[22] A.T.A. Gomes, S. Colcher, L.F.G. Soares, Towards a descriptive approach to model adaptable communication environments, in: Proceedings of the 1st International Conference on Networking (ICN'01), Colmar, France, Lecture Notes in Computer Science 2094 (2) (2001) 867-876.

[23] G. Hjálmtýsson, The Pronto platform – A flexible tookit for programming networks using a commodity operating system, in: Proceedings of the 3rd IEEE Conference on Open Architectures and Network Programming (OPENARCH'00), Tel Aviv, Israel, March 2000.

[24] C.A.R. Hoare, Communicating sequential processes, Communications of the ACM 21 (8) (1978) 666-677.

[25] Intel Corporation, IXA network processors. Available from <http://www.intel.com>.

[26] International Business Machines, PowerNP network processors. Available from <http://www.ibm.com>.

[27] International Telecommunications Union, ITU-T Recommendation Q.700: Introduction to CCITT Signalling System no. 7, ITU-T, 1993.

[28] International Telecommunications Union, ITU-T Recommendation Z.100: Specification and description language SDL, ITU-T, 1987.

[29] Institute for Software Research at the University of California, ArchStudio 3. Available from <www.isr.uci.edu/projects/archstudio>.

[30] Institute for Software Research at the University of California, xArch. Available from <www.isr.uci.edu/projects/xarch>.

[31] R. Isaacs, I. Leslie, Support for resource-assure and dynamic virtual private networks, IEEE Journal on Selected Areas in Communications (Special Issue on Active and Programmable Networks) 19 (3) (2001) 460-472.

[32] V. Issarny, C. Bidan, Aster: A framework for sound customization of distributed runtime systems, in: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96), Hong Kong, May 1996, pp 586-593.

[33] A. Joolia, G. Coulson, G.S. Blair, A.T.A. Gomes, K. Lee, J. Ueyama, Flexible programmable networking: A reflective, component-based approach, in: Proceedings of 4th Annual Postgraduate Symposium: The Convergence of Telecommunications, Networking and Broadcasting (PGNet'03), Liverpool, UK, June 2003.

[34] S. Karlin, L. Peterson, VERA: An extensible router architecture, in: Proceedings of the 4th IEEE Conference on Open Architectures and Network Programming (OPENARCH'01), Anchorage AK, USA, April 2001, pp 3-14.

[35] G. Kiczales, J. des Rivières, D.G. Bobrow, The art of the metaobject protocol, MIT Press, 1991.

[36] L.S. Lima, A.T.A. Gomes, S. Colcher, L.F.G. Soares, Um framework para provisão de QoS em redes móveis sem fio (english version: A framework for QoS provision in wireless networks), in: Proceedings of the 21st Brazilian Symposium on Computer Networks (SBRC'03), Natal, Brazil, May 2003.

[37] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using Rapide, in: IEEE Transactions on Software Engineering 21 (4) (1995) 336-355.

[38] P. Maes, Concepts and experiments in computational reflection, in: Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA'87), ACM Press, October 1987, pp 147-155.

[39] J. Magee, J. Kramer, Dynamic structure in software architectures, in: Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco CA, USA, ACM SIGSOFT Software Engineering Notes 21 (6) (1996) 3-14.

[40] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70-93.

[41] Microsoft Corporation, Component Object Model (COM). Available from <http://www.microsoft.com>.

[42] R.T. Monroe, A. Kompanek, R.E. Melton, D. Garlan, Architectural styles, design patterns, and objects, IEEE Software 14 (1) (1997) 43-52.

[43] R.S. Moreira, G.S. Blair, E. Carrapatoso, Constraining architectural reflection for safely managing adaptation, Proceedings of the IFIP/ ACM/ USENIX International Middleware Conference (Middleware'03) Workshops, Rio de Janeiro, Brazil, PUC-Rio, June 2003, pp 139-143.

[44] M.F. Moreno, A.T.A. Gomes, S. Colcher, L.F.G. Soares, Provisão de QoS adaptável em sistemas operacionais: O subsistema de rede (english version: Adaptable QoS provision in operating systems: The network subsystem), in: Proceedings of the 21st Brazilian Symposium on Computer Networks (SBRC'03), Natal, Brazil, May 2003.

[45] R. Morris, E. Kohler, J. Jannotti, M. Frans Kaashoek, The Click modular router, in: Proceedings of the 17th ACM Symposium on Operating Systems and Principles (SOSP'99), Kiawah Island SC, USA, December 1999, pp 217-231.

[46] O.T.J.D.D.L. Mota, A.T.A. Gomes, S. Colcher, L.F.G. Soares, Uma arquitetura adaptável para provisão de QoS na Internet (english version: An adaptable architecture for QoS provision in the Internet), in: Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC'01), Florianópolis, Brazil, May 2001.

[47] Object Management Group, CORBA component model specification, v3.0. Available from <http://www.omg.org>.

[48] N. Parlavantzas, G.S. Blair, G. Coulson, An approach to building reflective component-based middleware platforms, in: Proceedings of the 1st MSRC Summer Research Workshop, Cambridge, UK, September 2002.

[49] W. Pree, Design patterns for object-oriented software development, Addison-Wesley/ ACM Press, Reading MA, USA, 1995.

[50] A. Reid, M. Flatt, L. Stoller, J. Lepreau, E. Eide, Knit: Component composition for systems software, in: Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00), San Diego CA, USA, October 2000, pp 347-360.

[51] D. Schmidt, T. Suda, Transport system architecture services with high-performance communication systems, Journal of Selected Areas of Communications (Special Issue on Protocols for Gigabit Networks) 11 (4) (1993) 489-506.

[52] S. Schmid, LARA++ design specification, Technical report MPG-00-03, Computing Department, Lancaster University, Lancaster, UK, 2000.

[53] B. Schwartz, A.W. Jackson, T. Strayer, W. Zhou, R. Rockwell, C. Partridge, Smart packets for active networks, in: Proceedings of the 2nd IEEE Conference on Open Architectures and Network Programming (OPENARCH'99), New York, USA, March 1999.

[54] C.S. Soares-Neto, R.F. Rodrigues, L.F.G. Soares, (in press) Architectural description of QoS provisioning for multimedia application support, in: Proceedings of the 10th International Conference on Multimedia Modeling (MMM'04), Brisbane, Australia, January 2004.

[55] K. Sugiyama, K. Misue, Visualisation of structural information: Automatic drawing of compound digraphs, IEEE Transactions on Systems, Man and Cybernetics, 21 (4) (1991) 876-892.

[56] Sun Microsystems, Enterprise JavaBeans technology. Available from <http://java.sun.com>.

[57] C. Szyperski, Component software: Beyond object-oriented programming, 2nd ed., Addison-Wesley/ ACM Press, Reading MA, USA, 2002.

[58] A. van der Hoek, D. Heimbigner, A.L. Wolf, Software architecture, configuration management, and configurable distributed systems: A ménage a trois, Technical report CU-CS-849-98, Department of Computer Science, University of Colorado, Boulder, Colorado, USA, 1998.

[59] M. Ward, Proving program refinements and transformations, D.Phil. Thesis, St. Annes College, Oxford University, Oxford, UK, June 1989.

[60] D. Wetherall, J. Guttag, D. Tennenhouse, ANTS: A toolkit for building and dynamically deploying network protocols, in: Proceedings of the 1st IEEE Conference on Open Architectures and Network Programming (OPENARCH'98), San Francisco CA, USA, April 1998.

[61] S. Znaty, J-P. Hubaux, Telecommunication services engineering: Definitions, architectures and tools, in: Proceedings of the ACM/SIGPLAN European Conference for Object-oriented Programming (ECOOP'97) Workshops, Jyvaskyla, Finland, Lecture Notes in Computer Science 1357 (1997) 3-11.

[62] D. Batory, B.J. Geraci, Composition Validation and Subjectivity in GenVoca Generators, in: IEEE Transactions on Software Engineering 23 (2) (1997) 67-82.