Network Services on Service Extensible Routers*

Lukas Ruf, Károly Farkas, Hanspeter Hug, and Bernhard Plattner

Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology (ETH) Zurich
CH-8092 Zurich/Switzerland
{ruf,farkas,hhug,plattner}@tik.ee.ethz.ch

Abstract. Service creation on extensible routers requires a concise specification of component-based network services to be deployed and extended at node runtime. The specification method needs to cover the data-flow oriented nature of network services with service-internal control relations. Hence, it needs to provide the concept of functional service composition that hides the complexity of a distributed, dynamically code-extensible system.

We propose the PromethOS NP service model and its Service Programming Language to answer this challenge. They provide the concepts and methods to specify a network service as a graph of service chains with service components, and service-internal control relations. In this paper, we present the concepts of our service model, the syntax and semantics of its Service Programming Language, and demonstrate their applicability by an exemplary service specification.

1 Introduction and Motivation

One of the most significant problems of the Internet today is the lack of non-disruptive service creation and extension on demand on access and border routers. Non-disruptive service creation and extension have become key requirements due to the following trends:

– Function shift from the end-systems to the access networks: Function is moved from the end-systems towards the network to ease site operation, and to benefit from the economy of scale user-centric network services are deployed on access routers for the three plains [3] of data-path, control and management functionality. Examples are the protection of network sites [15] and the alleviation of network management and control [1].
– Router consolidation: For a reduction of costs of network management and operation, routers are consolidated. Larger devices, hence, are needed to satisfy the demands for the interconnection of different networks. Equipped with programmable network interfaces, these network nodes[1] provide suitable locations for new extended network services.

* This work is partially sponsored by the Swiss Federal Institute of Technology (ETH) Zurich, the Swiss Federal Office for Education and Science (BBW Grant 99.0533), and the Intel IXA University Program (Research Grant 22919). PromethOS v1 has been developed by ETH as a partner in IST Project FAIN (IST-1999-10561).

[1] We refer interchangeably to a router device by the term network node.

– Enabling Technologies: Network Processors [7,8] (NPs) appeared recently on the market. They provide generally an asymmetric programmable chip-multiprocessor architecture with functional units optimized for specific network operations. Usually, they are built of two different processor types, namely packet and control processors[2] (PPs and CPs respectively), that reside in a conceptual view at two different levels. CPs are built commonly of a general purpose processor (GPP) while PPs provide the architecture of a stripped down RISC processor supported by specialized co-processors to process packets at line-speed. Network interface cards with embedded NPs (so-called NP-blades) provide, thus, the required flexibility to extend large network nodes easily and increase the processing capacity of a network node simultaneously.

Router devices with NP-blades and multiple GPPs provide a powerful hardware platform. Service management and node operation, however, is complicated by the nature of these large heterogeneous network nodes. PromethOS NP [13,14] provides the flexible router platform that enables non-disruptive service creation and extension on any programmable processor at node run-time according to an extended plugin [5] model. For the definition of network services, however, a concise specification method is required. This specification method needs to export the capabilities of the underlying service infrastructure while it must abstract from the complexity of large multi-port router devices. Moreover, it must be general enough to cope with the large variety of network nodes. Thus, it needs to describe control and data relations among service components, and needs to specify additions to previously deployed network services with specific resource constraints in a flexible way.

Therefore, we present in this paper the PromethOS NP service model and propose its Service Programming Language (SPL) that is used to define and specify network services on the PromethOS NP router platform. The service model provides the concepts to deploy new network services composed of distributed service components on code-extensible routers and to extend previously deployed services with additional functionality. The SPL supplies the service programming interface of the router platform for the installation and basic configuration of new network services.

We structure the remainder of this paper as follows. In Sec. 2, we revise related work in the area of service models and specifications. Then, in Sec. 3, we introduce our service model and present our service programming language (SPL) with the definition of the relevant key productions. For proof of concept, we evaluate the SPL by its application on an exemplary service program in Sec. 4. The imaginary service program illustrates part of the capabilities and the flexibility of the SPL. In Sec. 5, this paper is then completed by a summary and conclusion followed by a brief outlook to further fields of application.

[2] NP vendors do not use a consistent naming scheme to refer to the code-extensible processors: the Intel IXP-architecture refers to the first-level processors as *microengines* while the IBM PowerNP identifies them as *picoprocessors* or *core language processors*. Second-level processors are named differently as well. For this reason, we refer to the first level of processing engines as *packet processors* and to those of the second level as *control processors*.

2 Related Work

Network service creation on active router platforms and deployment of services within the network have been a research area for quite a while. Research has been carried out on various levels of abstractions. We restrict our review of related work to four different projects in the area of service specification on active network nodes: Click [9] and NetScript [4] due to their service models and specification languages, Chameleon [2] due to its service model and process of service creation, and CORBA [11] because of its component model.

2.1 Click

The Click modular router [9] approach defines two environments of code execution (EE) on Linux: one is the in-kernel EE and the other is a Linux user space EE. Both support service creation according to a service specification of interconnected Click elements. Click elements provide the function of network services. Network services are defined by the specification of their inter-connection. Click uses so-called compound elements that allow for user-specified service class definition. A compound element consists of simple elements that specify the functions.

The Click specification language is a language that defines the elements and their inter-connection. Sub-classes of elements can be easily extended by own functionality, since new elements can be specified to create new functions in a C++-like style. The functions are then statically linked into the Linux kernel. Both, the in-kernel as well as the user space EE accept a Click service specification, resolve dependencies and create new network services.

While Click defines arbitrary service graphs by its specification language, the expressiveness to specify resource limits is not given. Moreover, its capabilities to extend previously deployed network services is not given following the architectural limitations of the Click EEs.

2.2 NetScript

NetScript [4] defines a framework for service composition in active networks that is programmed by three domain-specific languages: 1.) the dataflow composition language, 2.) the packet declaration language, and 3.) the rule-based packet classification language. The first defines a method to specify data path services as a composition of interconnected service components called boxes. The second is able to define the packet structure of network protocols, and the third defines the packet classification rules that are installed in the NetScript kernel. Boxes in NetScript provide a container for code or hardware-based service components, or other boxes in a recursive manner.

For our vision of service composition on a high-performance router, the first language, the dataflow composition language, is relevant. As a linear XML [16] specification of subsequent and interconnected datapath boxes that may be code or hardware elements, the NetScript dataflow composition language provides an interesting approach to our problem. However, it lacks the abilities to define control relations among control service components controlling other service components as well as for signalling conditions among subsequent service components. Moreover, the capabilities to extend previously deployed network services is not given, and it does not provide the expressiveness to specify resource and placement constraints of components.

2.3 Chameleon

Chameleon [2] is a node level service creation and deployment framework which provides an XML-based service description specifying a network service in an abstract way. The description is based on a recursive service model with containers, so-called abstract service components (ASCs). The ASCs group the functional entities and describe dependencies.

In Chameleon, service descriptions define network services as a composition of ASCs. A node local service creation engine (SCE) resolves the service description according to the local capabilities of the node into implementation service components and creates a tree like representation of them. These components are then deployed on the node by the help of a node configurator that provides the required interface towards the SCE to manage and control the platform.

Service components in Chameleon are modelled as functional entities supporting two different types of interfaces with push and pull call semantics for control and data path communication. Depending on the underlying NodeOS [10], Chameleon supports the interconnection of different EEs. In its current implementation, Chameleon makes use of a Click Linux kernel EE and a proprietary Java-based EE.

Chameleon focuses on the deployment of network services onto different network nodes. Thus, it provides the mechanisms and architecture to cope with a priori unknown network nodes. PromethOS NP, however, defines a specific architecture of a powerful router platform. Hence, for the modelling of network services, we need a service model that meets our needs and provides the capabilities to define the service infrastructure for heterogeneous NP-based network nodes, and, thus, resides at a different level of service modelling.

2.4 CORBA

CORBA [11] has defined the Common Object Request Brokerage Architecture to interconnect various, heterogeneous, distributed components by the mechanisms of the object request broker (ORB). The CORBA component model (CCM) [12] defines a component as a meta-type model with the encapsulated respective function. For component description and interface specification the Interface Description Language (IDL) is used. The CCM provides four different component interfaces named facets, receptacles, event sources and sinks. Facets are named interfaces for client interaction, receptacles are connection points, event sources are points that emit events to one or more interested event consumers, and event sinks are the corresponding event targets.

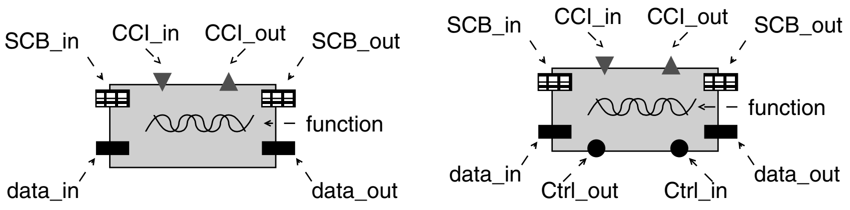
By the mechanisms of stubs and skeletons, a client-server architecture for distributed components is specified. The ORB provides the communication among distributed components in a way transparent to the creator of the CORBA service. Due to the level of abstraction, however, CORBA suffers from too much overhead for an efficient router platform.

3 Network Services

After the review of four specific projects of related work we introduce in this section our service model and then present its Service Programming Language (SPL) that is used to export the concepts of the service model.

3.1 Service Model

The goal of our service model is the modelling of a flexible service infrastructure that provides the mechanisms needed for the seamless integration of new service elements. Services are modelled as a graph of edges and vertices with edges representing chains of service components, and vertices defining the interconnection between them. The definition of network services is based on six constituent concepts: data path service components, control service components, service chains, guards, hooks, and name spaces that identify the service on an extensible router.



(a) Data Path Service Component (b) Control Service Component

Fig. 1. PromethOS NP Service Component

In Fig. 1, the models of a control and data path service component are visualized of which we refer to both by the term service component if no specific distinction is required. The service component defines a function according to the plugin model [5], but extends the interfaces provided. In addition to the data in- and output ports, our service component defines Service Control Bus (SCB) in- and output ports, and component control in- and output interfaces (CCIs). By the data in- and output ports, network traffic is received and sent out. The SCB serves for the propagation of service-internal signals between subsequent service components. The semantics of the signals on the SCB are service specific except for three signals (*ACCEPT*, *ABORT*, *CHAINEND*) that causes service infrastructure to accept a packet, abort the current service processing or signal the end of the service chain. The SCB interfaces allow for multiple read but only for a single write operation of the signal. Optionally, a service component exports control in- and output ports. CCIs provide the control interfaces to configure and retrieve control information of a service component at run-time.

Service components are defined for two different purposes: they provide data path service functionality (cf. Fig. 1(a)) or they provide service internal control functionality (see Fig. 1(b)). Control service components are either separate control components or they are inserted into the service path of data path service components, too. Control service components may be periodically triggered by timed events providing, thus, the required flexibility of control functionality. They offer the same interfaces but export in addition two controller interfaces (Ctrl_in and Ctrl_out in Fig. 1(b)) that define a multiplexing semantic to control multiple other service components provided the control service component implements the required functionality.

Both types of service components have specific resource requirements and characteristics. Resource requirements specify the amount of resources they need for their instantiation and their processing of network traffic while resource characteristics identify the type of resources needed. For example, different memory types exist on a NP-blade of which a service component consumes a specific amount or, as another example, different instruction set architectures (ISAs) are available on a NP depending on the processor cores implemented.

Service chains provide an aggregation of one or more service components that are strongly linked. A chain of strongly linked service components refers to the fact that only signals along the SCB are propagated between service components, and between service components and the service infrastructure. No demultiplexing of network traffic is available between the elements of a service chain allowing for fast pipeline-style processing by subsequent service components.

Guards provide the demultiplexing functions that control the acceptance of network traffic to enter service chains. Their definition has been inspired by the concept of guarded commands [6]. In our service model, guards are represented by service components that signal the acceptance or rejection of network traffic by the mechanisms of its SCB output port. Hence, they are the first service components of a service chain.

Hooks are key elements of the respective name space. Within a name space, they are identified by their label. They are created in the service program on demand. At creation time, the dispatching semantics are specified. If ingress hooks are created, they must be bound to a network interface. Otherwise, they must refer to previously created ones. Egress hooks may be dangling if required, thus implying the discard of arriving packets. The purpose of dangling outbound links is the provisioning of a hook for later service additions to extend provided functionality. Moreover, hooks serve for the embedding of service chains. They initiate and terminate a service chain. Multiple service chains are attached to hooks. Hooks provide the dispatching of network traffic to service chains since guards steer the demultiplexing of network traffic per service chain. Dispatching semantics have been defined by two different methods to which we refer by the terms *copy* and *first-match-first-consume*, respectively. The dispatching semantics are important for the specification of network services since it is a service design decision how network traffic is processed by different service chains.

We explain the difference between the two dispatching methods by the help of Fig. 2. In this figure, five service chains, enumerated from 1 to 5, are embedded between two

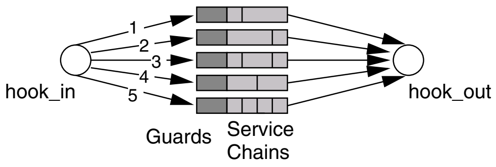


Fig. 2. Hooks, Guards and Service Chains

hooks labelled *hook_in* and *hook_out*. The order of service chains is defined by the service program created by the means of our SPL that is presented next. In case of the copy method, the initiating hook dispatches network traffic to all five service chains creating copies of the packets on acceptance by the guards. On the other hand, if the first-match-first-consume method has been specified, packets are presented to the guards in the order of service chain specification. Upon acceptance of a packet, the processing at hook_in is finished. For both methods, packets are discarded if no guard accepts a packet.

Name spaces are abstract constructions of our service model that are used to avoid name collisions between services. Name collisions would occur, for example, if hooks were labelled identically for different services and then reused for extending a previously deployed service with service additions.

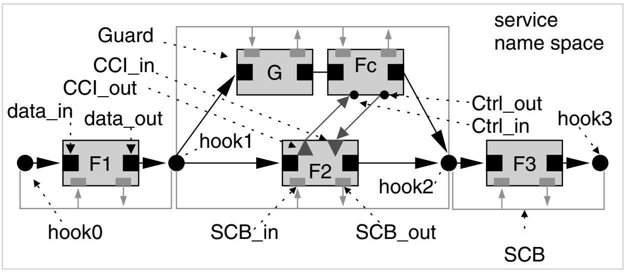


Fig. 3. Control and Data Path Relations Among Service Components

In Fig. 3, a service graph is presented that consists of four service components named $F1$, $F2$, $F3$ and F_c embedded between four hooks as well as of a guard labelled G that controls the packet acceptance for its service chain. It illustrates the data path and control relations between service components with F_c controlling $F2$. In Fig. 3, this controlling functionality is indicated by the letter $'C'$. Moreover, the figure visualizes the SCB that accompanies service chains.

3.2 The Service Programming Language

The SPL specifies a network service for the service infrastructure of the PromethOS NP router platform. It defines the Service Programming Interface (SPI) exported by the router platform for the creation and extension of new network services.

EBNF 1 presents the key elements[3] of the definition of the PromethOS NP *Service Programming Language* (SPL). The language definition is based on a modified form of the Extended Backus Naur Form (EBNF) [17] that deviates from Wirth's definition regarding the repetition-operator of elements denoted by bracelets ({..}). According to [17], the repetition-operator contains zero or more elements. For our purposes, we redefined the repetition-operator to produce *one* or more elements since

[3] Self-explanatory productions like, for example, BW, CYCLES or RAM are not included due to space constraints. Note that we refer to the $key = value$ pair by the term production.

```
ID            = "#" VALID_NAME.
TIMED         = "timed="DELAY.
BW_RES        = "bwmin="BW "bwmax="BW [ "pps="NUMBER ].
CPU_RES       = "cpumin="CPU "cpumax="CPU.
RAM_RES       = "type="ID "rammin="RAM "rammax="RAM.
PROC_TYPE     = ("ia32"|"ia64"|"np4"|"np4_pp"|"ixp2400"|"ixp2400_pp"|....).
CTRL_INFO     = (STRING | "file=" VALID_NAME ).
COMP_SPEC     = ( "src" [ ID ] | "bin" ( PROC_TYPE | ID ) )
                [ "|" CPU_RES ] [ { "|" RAM_RES } ].
COMP_IDENT    = ( [ "(" COMP_SPEC ")" ] VALID_NAME ID | ID ).
SERV_COMP     = COMP_IDENT [ ":" ID ] "(" [ CTRL_INFO ] ")".
CTRL_COMP     = [ TIMED ] SERV_COMP { "!" ID "@"NUMBER }.
CTRL_CHAIN    = "{" { CTRL_COMP } "}".
COMP_STRING=    "{" { SERV_COMP } "}".
GUARD         = "[" [ "|" BW_RES ] [ SERV_COMP ] "]".
HOOK_IN       = (ID | ">" ID [ "copy" ]  "?" INTF).
HOOK_OUT      = (ID | ">" ID [ "copy" ] [ "?" INTF ]).
SERV_CHAIN    = HOOK_IN
                 "@" [ TIMED ] [ GUARD ] COMP_STRING "@"
                HOOK_OUT.
SERVICE       = "{" ID [ "!" CTRL_CHAIN ] { SERV_CHAIN } "}".
```

EBNF 1. The PromethOS NP Service Programming Language

the optionality-operator is defined already by pairs of brackets ([..]). Thus, the semantics of the original zero-to-many repetition-operator is expressed as [{..}] by our EBNF variant.

The fundamental concept of the SPL is the linear specification of arbitrary service graphs consisting of service and control chains. Based on the concept of hooks to which service chains are attached, graphs are created out of the linear specification.

The key element of the SPL is the service component specified by the *SERV_COMP* production. It starts with the component identifier *COMP_IDENT*. Part of the component identifier is the specification of the resources (*COMP_SPEC*) required for its instantiation and the data format of the component. If it is specified as a reference to source code file (*src*), the platform assumes a component for the PromethOS NP processing environment for GPPs [13], and creates the respective binary component. Otherwise, in case of a binary component specification (*bin*), the SPL demands for the definition of the processor core type. This specification is relevant since different, incompatible ISAs may be available on a node. In both cases, the processor core can be specified (*ID*) on which the service component must be installed. This ID identifies a particular core per processor, and is required, for example, if not all processor cores are able to access particular hardware accelerators. The service component is then identified by the name of an object followed by its component instance identifier (*ID*). In case a service component instance is reused, the *ID* of a previously created instance is defined in the string of components. The router platform provides three pseudo components named *NIL*, *DROP* and *CLASSIFY* that exploit respective platform internal capabilities. Conceptually, they provide the interfaces like other service components, and their instances are identified by the same methods. Service components export CCIs optionally. In the SPL, they are specified by the " : *ID*" term. Control information (*CTRL_INFO*) to initialize a service component at service configuration time is specified then. It represents either a string of ASCII[4] characters or by a reference to an arbitrary object.

Control service components (*CTRL_COMP*) are service components that may be triggered by a timer event (*TIMED*), and that are bound to the control interfaces of other service components by the *!ID@NUMBER* statement. There, an *ID* references the control port exported by another service component, and *NUMBER* provides the control port multiplexing functionality needed to bind controlled service components to specific control mechanisms of a control component.

Guards are defined by the respective *GUARD* production according to its model introduced above. Please note that the specification of bandwidth limits and the maximal number of packets per second (*BW_RES*) are specified as part of the guard production since the dispatching function of hooks needs to control these limits already for the packet dispatching to guards such as to separate control from service function.

Hooks are specified by their respective productions (*HOOK_IN* and *HOOK_OUT*). Reuse of a hook is specified by the notion of a previously created hook identifier (*ID*). The creation of hooks is initiated by the literal " $>$ ", followed by the hook identifier (*ID*), the optional specification of the "*copy*" method for the hook's dispatching semantics and the binding of a hook to an interface "*?INTF*". In case no "*copy*" method is specified at hook creation time, the dispatching semantics follow the first-match-first-consume method. Note that the definition of the dispatching semantics for outbound hooks (*HOOK_OUT*) is needed since they are reused for further service chains potentially. Ingress hooks bound to network interfaces receive packets from the router platform following the copy method, i.e. all hooks bound to a network interface receive every incoming network packet. Analogous to the service components, the router platform provides a pseudo hook named *NIL* that is used to satisfy the SPL syntax for dangling hooks that are never extended, or for service chains that do not receive but just generate data.

Service chains are then specified by the *SERV_CHAIN* production that provides the aforementioned semantics of the service chain concept. Note the optional definition of a maximal delay (*TIMED*) the service chain is allowed to add on a packet processed by the service chain. The optional definition of the guard production allows for the specification of catch-all service chains as required for fall-back service paths if no previously defined guard accepted a packet.

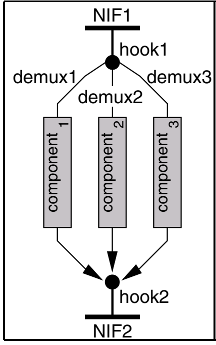
The service (*SERVICE*) is identified by its service identifier (*ID*) that specifies the service name space. Optionally, a service consists of a control chain (*CTRL_CHAIN*) that contains the control service components, followed by the definition of the constituent service chains for data path packet processing.

4 Evaluation

For proof of concept of our SPL, we illustrate its capabilities by a service program and its corresponding visualization hereafter.

[4] ASCII – American Standard Code for Information Interchange as defined by the ISO/IEC standard 646.

Table 1. Three Parallel Service Chains

Visualization	Chain 1	Chain 2	Chain 3
 NIF1 hook1 demux1 demux2 demux3 component 1 component 2 component 3 hook2 NIF2	{ #threeparallel > #hook1 ? NIF1 @/* HOOK */ [/*DEMUX1*/] { /*COMP_STRING*/ (bin ia32) component1 #instance1ID (/*CTRL_INFO*/) } @/* HOOK */ > #hook2 ? NIF2	/* extend hook1 */ #hook1 @/* HOOK */ [/*DEMUX2*/] {/*COMP_STRING*/ (bin ia32) component2 #instance2ID (/*CTRL_INFO*/) } @/* HOOK */ #hook2	/* extend hook1 */ #hook1 @/* HOOK */ [/*DEMUX3*/] { /*COMP_STRING*/ (bin ia32) component3 #instance3ID (/*CTRL_INFO*/) } @/* HOOK */ #hook2 }/* Service End*/

Three Parallel Service Chains. Table 1 presents a simple exemplary service program that defines a network service with three parallel service chains. The service program illustrates the linear specification of a service graph with parallel service chains. The service identifier (*#threeparallel*) is followed by the creation of hook1. No copy method is specified. Hence, its packet dispatching semantics follow the first-match-first-consume method in the top-down order of specified service chains. Hook1 is bound to one network interface (NIF) that is symbolized by the term *NIF1*. The service chain that consists of component1 is attached to hook1, first. While the figure in Table 1 illustrates the demultiplexing of flows to the particular service chains by attaching abstract demux conditions to the links between hook1 and the respective service chain, no real demultiplexing is specified in the service program. However, demultiplexing conditions are indicated in the service program by the respective comments. All service chains lead into hook2, which is bound to the second NIF (*NIF2*). The second and third service chains follow the same principle. Their specification differs from the first service chain by that hooks are re-used, i.e. the newly defined service chains are attached to the existing hooks.

5 Summary and Conclusions

In this paper, we have introduced the PromethOS NP service model and presented its Service Programming Language (SPL) that is used to specify network services. The SPL provides, hence, the Service Programming Interface (SPI) of PromethOS NP to create new network services and to define additions to previously deployed ones on extensible routers.

The service model provides the concept of a name space that is used to create the environment for network services of which multiple may reside in parallel on an extensible router platform like PromethOS NP. Within a name space, services are defined as a graph of service chains with constituent service components for data path processing. They are controlled by the service control chain realizing distributed, service internal control relations. Service chains are embedded between pairs of hooks. Hooks provide the dispatching functionality of network traffic to service chains that accept packets depending on their guards. Hooks are dynamically created within a service and serve from thereon as the reference point for service additions to extend previously deployed network services.

The SPL has been proposed as a context-free service programming language of our service model. Its syntax has been defined in a modified EBNF notation, and the semantics of the important production have been introduced extensively. For a proof of concept, we have applied our SPL to define an exemplary service program that illustrates the fundamental concept of a linear specification of arbitrary service graphs and their internal data path and control communications.

We are convinced that our service model with the SPL provide a suitable way to specify distributed network services for service extensible routers. The model contributes to research by three novelties: 1.) flexible service extensibility based on hooks that are dynamically created, 2.) the 1:n bi-directional control relation between a control and multiple controlled service components, and 3.) the service control bus that propagates signals between subsequent data path service components. The SPL proposes a concise method to specify network services that are based on our service model. Our SPL extends previous work by the concepts resource constraints assigned to service chains. The definition of the pseudo component NIL provides the methods to define syntactically correct service programs with cut-through channels, the DROP element supports explicit packet dropping, and the CLASSIFY component is used to exploit platform internal classification mechanisms like hardware supported packet classification engines. Moreover, the CLASSIFY component together with the instance re-use method provides the capability to exploit mechanisms of advanced network processors in which multiple disjoint rules may be compiled into an advanced matrix-based packet classification that all lead to the same service component.

Based on our service model, the service infrastructure of the PromethOS NP router platform has been designed and implemented. The SPL is currently used as the SPI to the PromethOS NP router platform for service creation and extension. However, we are convinced that our service model with its SPL provides the concepts for applications in a larger scope than only for node-local network service creation. As an example, we envision their use for other distributed component-based data processing applications, such as staged image processing that need service internal data and control relations.

References

1. Becker, T., Bossardt, M., Denazis, S., Dittrich, J., Guo, H., Karetsos, G., Takada, O., Tan, A.: Enabling customer oriented service provisioning by flexible code and resource management in active and programmable networks. In: IEEE International Conference on Telecommunications (ICT), Bucharest, Romania. IEEE, Los Alamitos (2001)
2. Bossardt, M., Antink, R.H., Moser, A., Plattner, B.: Chameleon: Realizing automatic service composition for extensible active routers. In: Wakamiya, N., Solarski, M., Sterbenz, J.P.G. (eds.) IWAN 2003. LNCS, vol. 2982. Springer, Heidelberg (2004)
3. The FAIN Consortium. D14: Overview FAIN Programmable Network and Management Architecture (May 2003)
4. da Silva, S., Florissi, D., Yemini, Y.: Composing active services with NetScript. In: Proc. DARPA Active Networks Worshop, Tucson, AZ (March 1998)

5. Decasper, D., Dittia, Z., Parulkar, G., Plattner, B.: Router Plugins: A Software Architecture for Next Generation Routers. In: Proc. of the ACM SIGCOMM 1998 Conf., Vancouver, British Columbia, Canada. ACM Press, New York (1998)
6. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM 18(8) (1975)
7. IBM Corp. Datasheet IBM NP4GS3(March 2004), http://www.ibm.com
8. Intel Corp. Intel IXP2xxx hardware reference manual (2003), http://www.intel.com
9. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M., Modular, C.: The Click Modular Router. ACM Transactions on Computer Systems 18(3) (August 2000)
10. Peterson, L. (ed.): NodeOS Interface Specification. Active Network Working Group (January 2001)
11. Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. TC Document 91.12.1, Revision 1.1, OMG (December 1991)
12. Object Management Group (OMG). CORBA Components. Technical Report Version 3.0, OMG (June 2002)
13. Ruf, L., Keller, R., Plattner, B.: A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors. In: Proc. of 2004 ACS/IEEE Int. Conf. on Pervasive Services (ICPS 2004), Beirut, Lebanon. IEEE, Los Alamitos (2004)
14. Ruf, L., Pletka, R., Erni, P., Droz, P., Plattner, B.: Towards High-performance Active Networking. In: Wakamiya, N., Solarski, M., Sterbenz, J.P.G. (eds.) IWAN 2003. LNCS, vol. 2982. Springer, Heidelberg (2004)
15. Ruf, L., Wagner, A., Farkas, K., Plattner, B.: A Detection and Filter System for Use Against Large-Scale DDoS Attacks in the Internet Backbone. In: Minden, G.J., Calvert, K.L., Solarski, M., Yamamoto, M. (eds.) Active Networks. LNCS, vol. 3912, pp. 169–187. Springer, Heidelberg (2007)
16. W3C XML Working Group. Extensible Markup Language (XML). Recommendation 6, W3C (October 2000), http://www.w3c.org
17. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Communication of the ACM 20 (1977)