Towards Router Specification in Curry: The Language ROSE *

J. Guadalupe Ramos[1], Josep Silva[2], and Germán Vidal[2]

[1] Instituto Tecnológico de La Piedad
Av. Tecnológico 2000, Meseta los Laureles, La Piedad, Mich., México
guadalupe@dsic.upv.es
[2] DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain
{jsilva,gvidal}@dsic.upv.es

Abstract. The development of modern routers require a significant effort to be designed, built, and verified. While hardware routers are faster, they are difficult to configure and maintain. Software routers, on the other hand, are slower but much more flexible, easier to configure and maintain, less expensive, etc. Recently, a modular architecture and toolkit for building software routers and other packet processors has been introduced: the Click system. It includes a specification language with features for declaring and connecting router elements and for designing abstractions.
In this work, we introduce the domain-specific language Rose for the specification of software routers. Rose is embedded in Curry, a modern declarative multi-paradigm language. An advantage of this approach is that we have available a framework where router specifications can be transformed, optimized, verified, etc., by using a number of existing formal techniques already developed for Curry programs. Furthermore, we show that the features of Curry are particularly useful to specify router configurations with a high-level of abstraction. Our first experiments point out that the proposed methodology is both useful and practical.

Key words: software engineering, declarative multi-paradigm programming, specification, routers.

1 Introduction

In heterogeneous environments, special devices to interconnect different technologies are often required. Within Internet networks, the *router* is that device. Basically, routers connect two or more networks and forward data packets between them. Thus, the primary function of a router is to determine the best path in a complex network. Originally, routers have been developed entirely as hardware components. However, there is a clear trend towards extending the set of functions that network routers should support. These new functions include,

* This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the Generalitat Valenciana under grant CTIDIA/2002/205, and by the MCYT under grants HA2001-0059, HU2001-0019 and HI2000-0161.

e.g., packet filtering, address translation, run proxies, performance monitoring, etc. The flexibility required to cope with all these new functions motivated the definition of so called *extensible routers* [7], which support run-time customization of router functionality.

The most important approaches to extensible routers are Scout, Router Plugins, and Click. In Scout [16], the basic abstraction unit is the *path*: a linear flow of data that starts at a source device and ends at a destination device. Each path is composed by *stages* that are instances from a specific *module*, which implements a well understood protocol (IP, TCP, etc). Scout provides tools to create, modify, schedule, and control paths. Implemented in the NetBSD operating system, Router Plugins allows users to write (limited) extensions to an IP router. These extensions can be placed at well known points—called *gates*—of the router's IP execution. Gates have been chosen to suit a wide variety of applications, like routing, packet scheduling, and security processing. Finally, Click [13, 12] is based on composing many simple *elements* to produce a system that implements the desired behavior. Each element may have multiple *ports* to connect it to other elements. These elements control every aspect of the behavior of the router, from communicating with devices to packet modification to queuing, dropping policies and packet scheduling. New configurations can be built by gluing elements together with a simple language (which is also called Click).

According to [7], Click is the most flexible of the three architectures above. Its ability to form virtually any configuration from the set of elements gives the programmer a high degree of freedom to modify routers incrementally and to add new services. On the negative side, this flexibility also implies that Click provides very little guidance on what constitutes a well-formed and meaningful configuration. For this purpose, Click already includes a set of tools which help the user to deal with complex configurations. Nevertheless, the inclusion of *semantic* aspects in Click (e.g., an abstract description of each element, with the number of input and output ports, how it modifies the passing packets, etc) would be very useful for the programmer.

In this paper we introduce Rose, a domain-specific language—which is based on Click—for router specification. Rose is *embedded* in Curry [11], a declarative multi-paradigm language which integrates features from the most popular declarative paradigms (namely, functional, logic and concurrent programming). Therefore, router configurations are first-class objects in Curry, which allows us to use the higher-level facilities of Curry, such as higher-order combinators, constraints, laziness, logical variables, etc. Moreover, there already exists a number of tools for transforming, optimizing, and verifying Curry programs—with a solid theoretical basis—that are also available to the programmer. Let us clarify that we do not intend to compete with Click. Rather, our aim is to develop a complementary approach. For instance, one can use Rose during the first design phases and, then, (automatically) translate the developed specification into Click. Furthermore, one can translate an existing Click configuration into Rose to perform some transformation and analyses and, then, translate the result back into Click. In other words, we do not plan to develop software routers in Curry (as Click does), only their semantic *specification*. Consequently, Rose elements only contain high level information which is useful for analysis, simulation, optimization, verification, etc. The main advantage of our proposal is that it provides an appropriate basis to develop specific analysis and optimization tools (while only a few such tools already exists for Click and, moreover, they have not been formally verified).

This paper is organized as follows. Section 2 introduces an overview of the language Curry. In Section 3, we present our approach to router specification: Section 3.1 reviews the Click language, Section 3.2 presents the specification of Click elements in Rose, Section 3.3 defines some composition operators to build larger components, and Section 3.4 briefly describes the implementation of Rose. Finally, Section 4 discusses some related works and Section 5 concludes and presents some directions for future work.

2 The Curry language

Curry [11] is a declarative multi-paradigm language which combines in a seamless way features from functional programming (nested expressions, higher-order functions, lazy evaluation), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). The development of Curry is an international initiative intended to provide a common platform for the research, teaching and application of integrated functional logic languages.

In Curry, functions are defined by a sequence of rules (or equations) of the form

$$f\ t_1 \ldots t_n\ \texttt{=}\ e$$

where $t_1, \ldots, t_n$ are *constructor* terms and the right-hand side e is an *expression*. The left-hand side must not contain multiple occurrences of the same variable. Constructor terms may contain variables and constructor symbols, i.e., symbols which are not defined by the program rules. Functions can be also defined by *conditional equations* which have the form

$$f\ t_1 \ldots t_n\ \texttt{|}\ c\ \texttt{=}\ e$$

where the condition (or *guard*) c can be either a Boolean function or a constraint. Elementary constraints are `success`, which is always satisfied, and *equational constraints* e_1 `=:=` e_2 between two expressions. The latter is satisfied if both expressions are reducible to a same ground constructor term (i.e., the so-called *strict equality* [6, 15]). Operationally, an equational constraint e_1 `=:=` e_2 is solved by evaluating e_1 and e_2 to unifiable constructor terms.

We can define *non-deterministic* functions either by providing several rules with overlapping left-hand sides or by introducing *free variables* (i.e., variables that do not occur in the left-hand side) in the condition or in the right-hand side of the rules. For instance, the following function non-deterministically inserts an element in a list (where `[]` denotes the empty list and `x:xs` a list with first element `x` and tail `xs`):

```
insert x []     = [x]
insert x (y:ys) = x : y : ys
insert x (y:ys) = y : insert x ys
```

Local declarations can be defined by using the `let` or `where` constructs. The following Curry function splits a list into two lists containing the smaller and larger elements:

```
split e []               = ([], [])
split e (x:xs) | e >= x  = (x:l, r)
               | e < x   = (l, x:r)
               where (l,r) = split e xs
```

Higher-order features include partial function applications and lambda abstractions. Function application is denoted by juxtaposition of the function and its argument. The evaluation of higher-order calls containing free variables as functions is not allowed (i.e., such calls are suspended to avoid the use of higher-order unification [10]). For instance, the well-known function `map` is defined in Curry by

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Lambda abstractions—anonymous functions—are expressions of the form

$\backslash \mathtt{x}_1 \ldots \mathtt{x}_n$ `->` *exp*

The application of the above lambda abstraction to input arguments $e_1, \ldots, e_n$ produces the same result as the function call "`foo` $e_1 \ldots e_n$", where `foo` is defined as follows:

`foo` $\mathtt{x}_1$ ... $\mathtt{x}_n$ `=` *exp*

Curry also allows the use of functions which are not defined in the user's program (*external* functions), like arithmetic operators, usual higher-order functions (`map`, `foldr`, etc.), basic input/output facilities, etc.

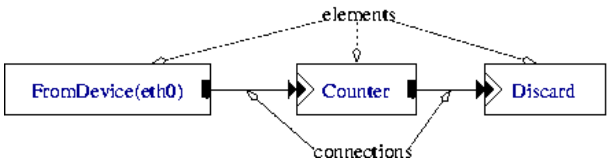
We refer the interested reader to the report on the Curry language [11] for a detailed description of all the features of the multi-paradigm language Curry.

3 Specifying Click Routers

In this section, we recall the fundamentals of the Click language and introduce the domain-specific language Rose to represent Click router configurations.

3.1 The Click Language

The Click programming language textually describes Click router configurations. Click has only two basic constructs which are enough to describe any configuration graph [12]: *declarations* create (instances of) elements and *connections* connect existing elements together. In order to build a router configuration, the user chooses a collection of *elements* and connects them via *ports* into a directed graph. For instance, the following graph shows several elements connected into a simple router that counts incoming packets and, then, throws them all away:



In the Click language, this router is specified as follows:

```
// Declarations                   // Connections
src  :: FromDevice(eth0);         src -> ctr;
ctr  :: Counter;                  ctr -> sink;
sink :: Discard;
```

Basically, Click elements [12] fall into one of the following categories:[3]

Packet sources. They spontaneously generate packets, either by reading them from the network, reading them from a dump file, creating them from specified data, or creating them from random data. They have one output and no inputs.

Packet sinks. They remove packets from the specified system, either by simply dropping them, sending them to the network, writing their contents to a dump file, or sending them to the Linux networking stack. They have one input and no outputs.

Packet modifiers. They are used to change packet data. They have one input and one (or two) outputs. Packets arriving on the input are modified and then emitted on the first output (the second output, if present, is for erroneous packets).

Routing elements. They choose where incoming packets should go based on a packet-independent switching algorithm, general characteristics of packet flow, or an examination of packet contents. For instance, a typical round-robin switch pushes each arriving packet to one of the outputs; the next packet will be pushed to the following output in round-robin order. They have one input and two or more outputs.

Scheduling elements. They choose packets from one of several possible packet sources. A Click packet scheduler is naturally implemented as an element with two or more inputs and one output. For instance, this element could react to requests for packets by choosing one of its inputs in turn until one produces a packet, pulling a packet from it, and returning that packet. When the next request is produced, it starts from the input after the one that last produced a packet. This amounts to a round-robin scheduler.

In the next section, we will illustrate the specification of the different Click elements in Rose.

[3] Let us note that only router elements are *visible*, e.g., when a packet is sent to the network it disappears from the specified system.

3.2 The Specification Language Rose: Basic Elements

Designing a new programming language implies a considerable amount of work: the definition of its syntax and semantics, the implementation of interpreters and compilers, the development of a number of useful tools for analysis, debugging, optimization, program manipulation, etc. An alternative approach is to define an *embedded* language, i.e., to develop libraries in some existing language—the *host* language—which allow us to define programs in the new language as *objects* in the host language. In this way, we can reuse the syntax, language features and tools of the host language.

In the following, we present the language Rose for the specification of Click routers. Since we do not want to define yet another specification language, we build Rose as an embedding in the multi-paradigm language Curry so that we can reuse the features and tools of a powerful declarative language. We use Curry because it has many useful properties for specification: the embedding is easy,[4] there are many features which are useful to describe routers (e.g., laziness, guarded rules, and higher-order combinators), and it allows the use of logical features (like logical variables, non-determinism and search), which can be useful for simulation and testing (see below).

Let us summarize the main advantages of our approach and, in particular, of embedding Rose in the multi-paradigm language Curry:

Data structures and recursion. In order to describe elements that have multiple inputs we use Curry lists. Often such elements can be defined for any size, a property that is called *genericity* in hardware description languages like VHDL. A common way of defining and manipulating such elements is to use recursion.

Polymorphism. Some elements can accept as parameters different types in such a way that they can be reused in many parts of the router with different input data.

Higher-Order Functions. Higher-order facilities are essential in order to build a router configuration by connecting or composing existing elements. The *composition* operators that will be described in Section 3.3 make extensive use of higher-order functions.

Laziness. Curry follows a *lazy* evaluation model, which means that functions are evaluated *on demand*. This is particularly useful to deal with infinite data structures (like packet streams). For instance, we can define a router as a function which generates an infinite stream of packets; however, if the user only demands the visualization of the first 10 packets, the remaining packets are not actually built.

Type System. Curry includes a standard type inference algorithm during compilation. Type errors can be very useful to detect errors in a router configuration, e.g., to detect that two incompatible elements have been erroneously connected.

[4] Indeed, Curry has already been used to embed other languages, e.g., a language for distributed programming [8].

Logic Features. Current proposals, particularly Lava [3] and Hawk [14], are based on pure functional languages (i.e., Haskell). In contrast, Curry provides logical features like non-deterministic functions, logical variables, built-in search, etc. Although these features have not been exploited yet, they will be useful to perform simulations with incomplete information (where the *holes* are represented by logical variables), to define some kinds of analysis, etc. This is subject of ongoing work and justifies our choice of Curry to embed the specification language Rose.

Click elements are packet processors. Therefore, each element in the router configuration can be abstracted by a function which takes a number of packet streams and returns a number of packet streams. In particular, each input port receives a packet stream and each output port emits a packet stream. Packets and streams are specified by means of the following types:

```
type Packet = [Int]
type Stream = [Packet]
```

A packet is regarded as a (finite) list of bytes, here encoded as integers, and streams are (potentially infinite) lists of packets. Typically, a Click element has the following type:

```
element :: [Conf] -> [Stream] -> [Stream]
```

i.e., it takes some configuration parameters (if any), a list of input streams (which can be empty if the element is a packet source), and returns a list of output streams (which can be empty if the element is a packet sink). Configuration parameters contain additional information to define the particular element behaviour. Usually, they fit into one of a small set of data types: IP addresses, for example, or integers, or lists of IP addresses. Our definition of this data type has the following form:

```
data Conf = I Int     --an integer (used, e.g., in 'strip')
          | Eth Int   --network devices (like eth0, eth1, ...)
          | Pat [Int] --patterns (used, e.g., in 'classifier')
          | Ip [Int]  --an IP address
          ...
```

Now we show examples of the five kinds of elements we distinguished in the previous section.

Packet sources. These elements are encoded as functions with no input streams and only one output stream. For instance, the Click element `FromDevice(eth)` is a packet source, where `eth` is a network device. It is specified in Rose as follows:

```
fromDevice :: [Conf] -> [Stream] -> [Stream]
fromDevice [eth] [] = infiniteSource [eth] []
```

where the auxiliary function `infiniteSource` is similar to `fromDevice` but *simulates* the stream of packets coming from a given network address. Clearly, there is no difference in Rose between the Click elements `fromDevice` and `infiniteSource`, since we do not implement actual connections to the network.

Packet sinks. These elements are encoded as functions that return the input streams *with no modification* (in contrast to Click packet sinks, which have no output streams). The reason for this behavior is to allow the user to *observe* the output of the router, which can be useful during the specification phase to test the router behavior. For instance, the Click element `ToDevice(eth)` sends a stream of packets to some network device. In Rose, we specify this element as follows:[5]

```
toDevice [eth] [ps] = [aux ps]
                      where aux []     = []
                            aux (p:ps) = p : aux ps
```

Here, we discard the network device in the local function `aux` since we are not interested in the network devices to which packets are sent. If more detailed information would be needed, we could easily modify the above function in order to return pairs (ethernet device, packet).

Packet modifiers. In contrast to Click, we consider packet modifiers with one input stream and one output stream (i.e., we do not consider erroneous packets). As an example of a packet modifier, we show the specification of the Click element `Strip(n)`, which deletes the first `n` bytes from each packet (e.g., to get rid of the Ethernet header). In Rose, it is encoded as follows:

```
strip [n] [ps] = [st_ n ps]
                 where st_ m []     = []
                       st_ m (q:qs) = drop m q : st_ m qs
```

where `drop` is a predefined Curry function—it is part of the Curry *prelude*—which returns a suffix of a given list without the first `n` elements.

Routing elements. These elements have only one input stream and two or more output streams. As an example, in the following we consider the Click element `Classifier(`$\texttt{pat}_1$`,...,`$\texttt{pat}_n$`)` which classifies packets according to their contents. To be more precise, it takes a sequence of patterns $\texttt{pat}_1,\ldots,\texttt{pat}_n$ which are pairs `offset/value`, compares each incoming packet data against the set of patterns, and sends the packet to an output port corresponding to the first pattern that matched.

For instance, the Click declaration

```
Classifier(12/0806 20/0001,
           12/0800,
           -);
```

[5] In the following, we do not show the type of elements since it always has the form "`[Conf] -> [Stream] -> [Stream]`".

creates an element with three outputs intended to process Ethernet packets: ARP requests (offset 12 has the hexadecimal value 0806 and offset 20 has the hexadecimal value 0001) are sent to the first output, IP packets (offset 12 has the hexadecimal value 0800) are sent to the second output and all other packets (denoted by the special case "`-`") are sent to the third output.

In Rose, we denote patterns by a list of expressions "`Pat` *cons*", where *cons* is a list with an odd number of integers and such that each pair of integers n, m denotes the following condition: byte n of the packet must have value m. The last integer denotes the output stream (starting from zero). Also, the special case "`-`" (which matches every packet) is represented by a pattern with only one element, the output stream. In this way, the above configuration string, "`12/0806 20/0001, 12/0800, -`", is specified in Rose as follows:

```
[Pat [12,8,13,6,20,0,21,1,0], Pat [12,8,13,0,1], Pat [2]]
```

Here, we denote each Click pair `offset/value` by several pairs `byte,value`; for instance, the Click pattern 12/0800 is represented by the sequence $12, 8, 13, 0$.

Now, the Rose specification of the element classifier can be given as follows:

```
classifier pats [p:ps] = add n p qs
                         where n = class pats p
                               qs = classifier pats [ps]

class (Pat pat : pats) p = let n = class_ pat p
                           in if n == (-1) then class pats p
                                           else n

class_ [n] p = n
class_ (pos:val:es) p = if p!!pos == val then class_ es p
                                          else -1

add n p qs = if n==0 then (p : qs!!0) : tail qs
                     else (qs!!0) : add (n-1) p (tail qs)
```

In general, a call of the form `classifier pats (p:ps)` returns a list of streams `[qs`$_1$`, ...,p:qs`$_n$`,...,qs`$_k$`]`, where `[qs`$_1$`, ...,qs`$_n$`,...,qs`$_k$`]` is the list returned by `classifier pats ps`. Function `classifier` proceeds by first determining the output stream `n` of the first input packet `p`—using the function `class`—and then adding—using the function `add`—packet `p` on top of the `n` stream of the result of applying classifier recursively to the remaining input packets `ps`.

Here, given a call of the form "`ps!!n`", the predefined function `!!` returns the nth element of the list `ps`, while a call of the form "`tail ps`" returns the tail of `ps`.

Scheduling elements. These elements have one output stream and two or more input streams. They choose a packet from an input stream—according to some scheduling policy—and send it to the output stream. In our setting, we assume that there are always available packets in each input stream. For instance, the following Rose function implements a trivial round-robin strategy:

```
roundRobin [] ss = [rr ss]
   where rr []     = []
         rr (s:ss) = if s==[] then rr ss
                              else (s!!0) : rr (ss++[tail s])
```

Function `roundRobin` takes a list of input streams and no configuration parameters. It returns the packet on top of the first input stream, moves the remaining packets of this stream to the last position, and calls `roundRobin` recursively.

3.3 Composition Operators

Rose provides the user with a Curry library containing the specification of the main Click elements. However, this is not enough to specify a router configuration. The programmer also needs simple ways to connect single elements in order to build larger components. For this purpose, we also provide typical *composition operators*, which are defined by using the higher-order facilities of the language Curry.

The simplest composition operator is used to combine a number of elements in such a way that the output of one element is the input of the next one. This operator is defined as follows:

```
seq :: [[Stream] -> [Stream]] -> [Stream] -> [Stream]
seq []          = id
seq (elem : es) = \input -> seq es (elem input)
```

where `id` is the identity function. Function `seq` takes a list of elements and returns a new element whose input in the input of the first element in the list and whose output is the output of the last element in the list. For instance, the following function represents the router shown in Sect. 3.1:

```
simpleRouter = seq [fromDevice [Eth 0], Counter [], Discard []]
```

Another useful composition operator is `mult`, which is used to connect a routing element (with multiple output streams) to several packet modifiers. The specification of `mult` is as follows:

```
mult :: ([Stream] -> [Stream]) -> [[Stream] -> [Stream]]
        -> [Stream] -> [Stream]

mult elem es = \input -> mult_ es (elem input)

mult_ :: [[Stream] -> [Stream]] -> [Stream] -> [Stream]

mult_ es ss = concat (m es ss)
              where m [] []          = []
                    m (e:es) (s:ss) = (e [s]) : m es ss
```

where `concat` is a predefined Curry function to concatenate a list of lists. Function `mult` takes an element with n output streams and a list of elements with one input stream and connects them. It produces a new element whose input is the input of the single element and whose output streams are the output streams of the list of elements. For instance, the following function

```
classST = mult (classifier [Pat [12,8,13,6,20,0,21,1,0],
                            Pat [12,8,13,0,1],
                            Pat [2]])
               [strip [I 14], checkIPHeader [], toDevice [Eth 0]]
```

creates a new component that takes an input stream and sends ARP requests to element `strip` (which deletes the first 14 bytes from each packet), IP packets to element `checkIPHeader` (which checks that the packet length is reasonable and that the IP source address is a legal unicast address), and all other packets to the packet sink `toDevice [Eth 0]`.

There are more composition operators in Rose (e.g., to compose several packet modifiers with a scheduling element) that are not shown here.

3.4 Implementation

We have implemented a Curry library to design router specifications in Rose. This library contains

- a significant subset of the Click router elements,
- the basic composition operators, and
- some useful functions to test a router configuration.

Test functions have the following form:

```
test router n port = take n (router!!port)
```

In this way, one can obtain the first `n` packets sent to the output stream `port` by the router specification `router`, thus avoiding an infinite loop if the input stream is infinite. Predefined function `take` returns the `n` first elements of a given list. Additionally, the library contains a number of auxiliary functions, like common packet positions:

```
colAnnEth = 38             --color annotation in Ethernet packets
colAnnIP  = colAnnEth - 14 --color annotation in IP packets
ttlIP     = 8              --TTL field position in IP packets
...
```

or useful functions to manipulate packet data, like the application of a function to a given packet position:

```
appFun (p:ps) f n | n == 0  = f p : ps
                  | n > 0   = p : appFun ps f (n-1)
```

Moreover, in order to show the practicality of the ideas presented so far, we have specified a typical IP router (from [12]) in Rose.

Preliminary experiments are encouraging and point out the usefulness of our approach. More information is available at the following URL:

http://www.dsic.upv.es/~jsilva/routers/

4 Related Work

To the best of our knowledge, there is no previous approach for the specification of router configurations based on a well-established declarative language. We find, however, similar approaches for the design and verification of hardware components based on a language embedded in the lazy functional language Haskell [3, 14]. In [14], a domain-specific extension of the pure functional language Haskell—called Hawk—is introduced. Hawk was designed for the specification of microprocessors from a behavioral description approach. The main purpose of this language is to elevate the abstraction level of the specifications. In Hawk, a *signal* is an infinite stream of values $\langle x_0, x_1, x_2, \ldots \rangle$:

```
type Signal a = (Int -> a)
```

in such way that we can sample a signal `s` at a given clock cycle `n` by evaluating `s` applied to `n`. Then, circuits are defined as functions and, compositionally from them, complex circuits can be designed.

Another recent approach is the Lava language, introduced in Claessen's PhD thesis [3]. Lava has been successfully applied in industry, particularly in Xilinx Inc., for the development of high-speed digital signal processing circuits and filters for drawing Bezier curves. Lava is a Haskell embedded language that provides a set of libraries with many functions for the design of hardware circuits. In particular, Lava defines a signal as a list of Boolean values with a set of operators on that type. Circuits are defined as functions that work with signals, operators and other functions.

Clearly, our approach is inspired by the above works in that we also define an embedded language rather than a completely new language. We think that this approach greatly simplifies the development of new domain-specific languages. There are, however, some important differences:

– The base language is different. While [3, 14] embed their specification language in the lazy functional language Haskell, we embed Rose in the multi-paradigm language Curry, which extends Haskell with logical features and concurrent constraints. We think that these additional facilities can be useful for future developments: simulation, verification, optimization, etc.
– The specified systems are different. While [3, 14] specify hardware circuits, we deal with router configurations. Although they share some similarities, the specification of routers has some particularities that are not present in the specification of hardware circuits.

5 Conclusions

In this paper, we have introduced the domain-specific language Rose for the description of router configurations. It is embedded in the declarative multi-paradigm language Curry, which means that we have available all existing Curry tools for analysis, optimization, program manipulation, etc. For instance, we can use a Curry partial evaluator [1] to specialize generic router configurations, a fold/unfold transformation system [2] to manipulate router configurations in a semantics-preserving way, or use the tracing facilities of the PAKCS environment [9] to debug router specifications. We have shown how the basic Click elements are specified in Rose, as well as the main composition operators which are necessary to build larger configurations. In this way, Rose allows one to specify router configurations concisely, modularly and re-usably, while retaining its declarative nature.

There are many interesting topics for future work. On the one hand, we plan to extend Rose in order to cover all the existing Click elements. Also, it would be interesting to implement translators from Click into Rose and vice versa. This will facilitate the interaction with Click so that Rose can be regarded as a practical tool for the design and verification of Click router specifications. Finally, we will investigate the definition of specific analyses for Rose specifications, e.g., by using well-established abstract interpretation techniques [4, 5].

Acknowledgments

We would like to thank Eddie Kohler for useful comments and suggestions.

References

1. Albert, E., Hanus, M., Vidal, G.: A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. Journal of Functional and Logic Programming **2002** (2002)
2. Alpuente, M., Falaschi, M., Moreno, G., Vidal, G.: A Transformation System for Lazy Functional Logic Programs. In Middeldorp, A., Sato, T., eds.: Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming (FLOPS'99), Springer LNCS 1722 (1999) 147–162
3. Claessen, K.: Embedded Languages for Describing and Verifying Hardware. PhD thesis, Chalmers University of Technology and Gøteborg University, Department of Computing Science (2001)
4. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of Fourth ACM Symp. on Principles of Programming Languages. (1977) 238–252
5. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proc. of Sixth ACM Symp. on Principles of Programming Languages. (1979) 269–282
6. Giovannetti, E., Levi, G., Moiso, C., Palamidessi, C.: Kernel Leaf: A Logic plus Functional Language. Journal of Computer and System Sciences **42** (1991) 363–377
7. Gottlieb, Y., Peterson, L.: A Comparative Study of Extensible Routers. In: 2002 IEEE Open Architectures and Network Programming Proceedings. (2002) 51–62
8. Hanus, M.: Distributed Programming in a Multi-Paradigm Declarative Language. In: Proc. of the Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99), Springer LNCS 1702 (1999) 376–395
9. Hanus, M., Antoy, S., Koj, J., Sadre, R., Steiner, F.: PAKCS 1.5: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany (2003)

10. Hanus, M., Prehofer, C.: Higher-Order Narrowing with Definitional Trees. Journal of Functional Programming **9** (1999) 33–75
11. Hanus (ed.), M.: Curry: An Integrated Functional Logic Language. (Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`)
12. Kohler, E.: The Click Modular Router. PhD thesis, Massachusetts Institute of Technology (2001)
13. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.: The Click Modular Router. ACM Transactions on Computer Systems **18** (2000) 263–297
14. Matthews, J.: Algebraic Specification and Verification of Processor Microarchitectures. PhD thesis, University of Washington (2000)
15. Moreno-Navarro, J., Rodríguez-Artalejo, M.: Logic Programming with Functions and Predicates: The language Babel. Journal of Logic Programming **12** (1992) 191–224
16. Peterson, L., Karlin, S., Li, K.: OS Support for General-Purpose Routers. In: Workshop on Hot Topics in Operating Systems (Hot-OS-VII), IEEE Computer Society Technical Committee on Operating Systems (1999) 38–43