Model Checking Programmable Router Configurations

Luca Zanolin[1], Cecilia Mascolo[2], and Wolfgang Emmerich[2]

[1] Politecnico di Milano, Dipartimento di Elettronica ed Informazione
20100 Milano, Italy
Zanolin@elet.polimi.it
[2] Dept. of Computer Science, University College London
Gower Street, London WC1E 6BT, UK
{C.Mascolo , W.Emmerich} @cs.ucl.ac.uk

Abstract. Programmable networks offer the ability to customize router behaviour at run time, thus increasing flexibility of network administration. Programmable network routers are configured using domain-specific languages. The ability to evolve router programs dynamically creates potential for misconfigurations. By exploiting domain-specific abstractions, we are able to translate router configurations into Promela and validate them using the Spin model checker, thus providing reasoning support for our domain-specific language. To evaluate our approach we use our configuration language to express the IETF's Differentiated Services specification and show that industrial-sized DiffServ router configurations can be validated using Spin on a standard PC.

1 Introduction

Most routers in use in the current Internet are rather simple; they receive packets from one network interface, investigate the packet header that encodes the target IP address and forward them a router that is closer to the target. Most current routers implement the so-called best-effort model, which implies that all packets are *equal citizens*; they are all processed in the same manner. There are, however, applications, such as audio-conferencing or video-on-demand, that could be improved by quality of service (QoS) guarantees provided by the network. Moreover, there are different classes of users; companies might be prepared to pay a premium for performance and bandwidth guarantees. These guarantees cannot be given with the current best-effort model of the Internet.

To address this question, a relatively novel strand of network research investigates programmable networks [2], which give up the assumption that every network packet is handled in the same way by a router. Instead, the router executes a program that controls more intelligently how network packets are to be handled and forwarded to other networks. Such programmable routers can then be used to implement QoS Internet standards, such as the Differentiated Services model (DiffServ) [1]. DiffServ suggests marking packets in order to identify their service class at boundary routers so that then network traffic can be shaped by

delaying low class packets or even dropping them. The identification of service classes, conditions on shaping and dropping, and traffic management can then be implemented in router programs and thus different qualities of network services can be provided by a set of programmable routers.

The rise of programmable routers implies two interesting software engineering research questions. Firstly, router programs need to be changed on a regular basis in order to introduce new services without having to shut down the routers. Secondly, prior to updating a single router or a set of routers with a new router program, network administrators may want to check that their programs do not compromise network performance and reliability.

The main contribution of this paper is a solution to the second question. We describe a high-level router programming language that can be used to define the packet processing performed by a programmable router. We define the semantics of the language in an operational manner by mapping it to Promela, the specification language defined for Spin [8]. We then show how linear temporal logic (LTL) [13] can be used to specify safety and liveness properties for a router and demonstrate that Spin can efficiently and effectively check whether a given router program meets these properties. We discuss an evaluation of our approach using a set of DiffServ router programs and show how the model checking support is integrated into a network administration environment. By focusing on the software engineering aspect of this inter-disciplinary project, this paper presents an interesting case study for the systematic engineering of an application-specific configuration language whose definition has been inspired by graphical architecture description languages, the definition of its operational semantics and the provision of reasoning support using model checking techniques.

The paper is structured as follows. In Section 2, we briefly introduce the notion of programmable routers and describe our router configuration language. We define the operational semantics of the configuration language in Section 3 by mapping it to Promela. In Section 4 we show how we use that mapping to prove router properties with Spin. In Section 5, we evaluate our approach and in Section 6 we compare it to related work. Finally, in Section 7 we discuss future research directions of this project.

2 Programmable Routers

A programmable router can be configured in order to exploit the network resources according to different requirements. There are many applications of programmable routers, including firewalls that can rapidly respond to denial of service attacks, virtual private networks, traffic shaping, and provision of configurable quality of services.

We have implemented a programmable router, called `Promile` [12]. `Promile` is composed of two layers: the Router Kernel and the XML-bAsed Middleware (XAM). The kernel deals with packet forwarding, while XAM manages the router configuration. XAM presents the administrators with an abstraction of the routing machine, allowing them to modify the router to provide new services or mod-

ify old ones. We call the router abstraction *configuration*, because it configures how the router manages packets. The configuration is a high level abstraction, described by a set of *modules*.

When a packet is received by the router, it is given to a module, that, after applying a function, sends it to the next module. When the packet reaches the last module, the packet is re-injected into the network and forwarded to the next hop. According to its class and its header, a packet can go through different modules allowing the router to provide different services to different packet flows.

In order to implement qualities of service at the network level, the IETF has recommended the Differentiated Services (DiffServ) model [1]. In this paper, we use DiffServ to illustrate our approach to model checking programmable routers. DiffServ assigns a *class* to each packet and DiffServ routers handle and forward packets according to the class of the packet. The class of the packets is not used as a priority, but it identifies the type of service required for handling the packet.

Figure 1 is a simple example of a DiffServ router configuration. In this scenario, we suppose that the router is able to provide only two kinds of services: *video conference* and *file download*. Briefly, the first service has to guarantee the packet delivery with a given fixed bandwidth (the service is not interested into increasing throughput infinitely, as this would not improve the quality of a video conferencing application). The second service is interested in maximizing the packet transmission. As a consequence, we can delay packets from a video conference only if they are above the bandwidth requirement, while if the network is congested, we can delay or even drop the packets from a download session. Using a DiffServ architecture, each service is mapped into a different class, which is recorded in the IP packet header.

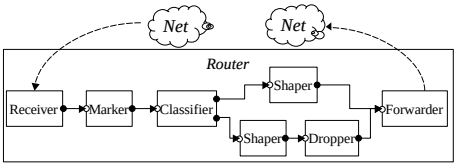

Fig. 1. Router Configuration

As shown in Figure 1, the *Receiver* module receives packets from the network and forwards them to the *Marker* module. According to the fields stored inside the packet header, the *Marker* assigns packets to a particular class. In this example, we can argue that *Marker* assigns the class according to the source host; we assume that *Marker* knows which hosts are downloading data and which are in a video conference session; obviously, this is a big simplification, but showing how to set up a DiffServ network is beyond the scope of this paper. More details are provided in [12].

When packets have been dealt with by *Marker*, they are sent to the *Classifier* module that, according to their class, routes them into the *upper* path, that is, the video conference/first class path, or into the *lower* path, that is, the download/second class path. When the packets are routed into the upper path, they go through the *Shaper* module, that is able to delay them. Otherwise, the packets can be routed into the lower path and go through the *Shaper* and *Dropper* modules where the packets can be delayed (*Shaper*) or even dropped (*Dropper*) in order to reduce network congestion. We now use this example to briefly introduce the concept of our application-specific language.

2.1 Modules

Modules are the basic building blocks of router configurations. By interconnecting modules and parameterizing their behaviour, we define the configuration of the router that provides the desired services. A module is a black box that applies a function to incoming packets. Modules communicate with other modules through *gates*. A packet is received from an input gate and sent through an output gate. A module can have multiple input gates in order to distinguish different kinds of packets. Most modules have both input and output gates, with two exceptions. A module with only output gates receives incoming packets from the network; this kind of module is called *source module* and usually describes an input network interface. Similarly, a module that only has input gates forwards packets into the network; this module is called *sink* and it models an output network interface. When a packet reaches the router, it is firstly managed by a source module and, after being processed by a configuration is sent to the next router or its destination by a sink module.

2.2 Rules

A module can be configured through a set of rules. According to these rules, the module applies its internal functionalities only to a sub-set of the incoming packets and it routes them through its output gates. A rule identifies packets through a set of fields related to the packet header and to the environment. The field sets may be different depending on the module's behaviour; this flexibility allows to precisely identify when a module functionality should be applied.

Num	**Source**		**Destination**		**Class**
	TCP	IP	TCP	IP	
1.	10	128.16.8.57	20	128.16.10.27	1
2.	10	128.16.10.27	*	*	2
3.	1000	*	1000	*	10
4.	1200	*	*	*	10

Table 1. Rules

Table 1 shows a sample rule set of the *Marker* module. It describes how classes (last column) are assigned to packets based on source and destination IP

addresses. For instance, the first rule identifies the packet class by specifying both source and destination packet address and assigns it to the *first class*. Using these rules, we can flexibly parameterise the behaviour of individual module instances.

2.3 Connections

Connections start from an output gate and lead into an input gate describing the packet flow among the modules. Connections can describe complex module graphs that should comply with the following properties. Firstly, any output gate must be connected to an input gate to fully specify the packet forwarding from module to module. Secondly, only one connection can start from an output gate. This ensures that the packet forwarding is deterministic. Finally, multiple connections may end in the same input gate and they may define elaborated router configuration with loops.

2.4 Configuration Update

The module instances, their interconnections, as well as the rules that have been used to parameterise the behaviour of module instances determine the *configuration* of a router. The management of these configurations for a `Promile` router is an important aim of the work described in this paper. The network administrator is able to change the router configuration without stopping it and, moreover, without any traffic disruption; as a consequence, the network can quickly react to changes in its environment, updating its configuration according to the application requirements.

When a network administrator, or an automatic tool, wants to update the router configuration, they have to deal only with the router abstraction provided by `Promile`. Prior to defining a new configuration, the administrator wants to be sure that the new configuration is meaningful. In order to do so, we have to support the definition of invariant properties.

2.5 Properties

The properties that network administrators might want to prove can be divided into three groups: *routing*, *service*, and *performance* properties.

Routing properties are concerned with the router and they define how the packet should be managed independently from provided services (i.e., *video conference support*); these properties guarantee that the router always works correctly and it is eventually able to handle new incoming packets. As these properties are not service-specific, they are almost the same for all the routers in the network and they are usually defined by the router developers. For instance, a routing property could be concerned with the module graph; as we have described, the module connections may be cyclic and this might cause packets to loop infinitely in the graph if this behaviour is not prevented by the configuration rules of modules contained in the cycle.

Service properties are concerned with the services that are to be implemented with a router configuration. With these properties, we want to guarantee that the configuration provides the services correctly. As these properties are service dependent, they can change from router configuration to router configuration. In general, they can be defined by administrators. For instance, an administrator can require that some functions are never applied to particular packets (i.e, never drop packets of premium users).

Performance properties are concerned with the router performance and they try to maximize router throughput through an optimisation of the configuration. Both service and routing properties are necessary, while performance properties are not vital. However, as optimal performance is an important feature for any router, we argue that a router has also to satisfy this kind of properties. Performance properties may be defined by the network administrator, but also by the router developer. A performance property, for example, may want to forbid that a packet is first marked and then dropped by the router, as this would make the router wasting time marking the packet that, instead, should have been dropped straight away.

Another performance property that should be checked is concerned with rules and modules. According to the router packet flow, some rules could be irrelevant, as no packets match them. In order to increase the router speed these should be removed. We should also check if all the modules are reachable by a packet, otherwise we should remove them.

3 Operational Semantics

In order to prove that the router satisfies desired properties, we translate the router configuration into a formal specification. We can then apply model checking techniques to validate the router configuration against the desired properties. By defining this translation, we give an operational semantics to the router configuration language.

We split this translation into two steps. The first step translates the router configuration into an *intermediate representation*, which describes each module through a set of *components*; the second step produces a Promela specification that defines the operational semantics and can be model checked by Spin. We assume that the `Promile` module types may be developed by third parties. In order to enable model checking of router configurations that instantiate such third party module types, module developers have to define the semantics of their module types. The first advantage of our two stage mapping is that we have defined the intermediate representation in such a way that it is relatively straightforward for a module developer to define the semantics of a module type using that intermediate representation.

We now describe the intermediate representation and its mapping to the Promela language.

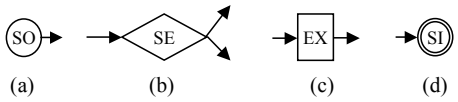

Fig. 2. Components

3.1 Intermediate representation

The intermediate representation supports four component types: *source* (SO), *selector* (SE), *executor* (EX), and *sink* (SI) (Figure 2). A *source* (Figure 2a) forwards new packets to the next component; this component does not actually create packets, but it directly takes them from the network. A *selector* (Figure 2b) receives packets and routes them to one of possibly several components according to selection rules that it applies. The selector is working as a *multiplexer*, where the rules describe how the incoming packet must be forwarded to the next component. A selector has one input and a list of numbered outputs; if a received packet matches no rules, it is sent through the default output (0). An *executor* (Figure 2c) receives packets and forwards them after applying a function. An executor applies the same function to all packets that it processes. A *sink* (Figure 2d) defines the end of the packet path. Arrows denote connections between components and they determine packet flows.

A DiffServ module of type *Dropper* could be mapped into the intermediate representation shown in Figure 3. The semantics of this module type is described using a selector component (SE) that chooses the packets that should be dropped and forwards them to the executor component (EX) that applies the module specific function (i.e., drops packets) and sends them to the sink component (SI) where the packet disappears and it is not forward any further.

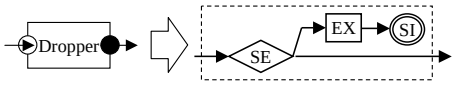

Fig. 3. Translation of a Dropper Module

The next step is to translate this intermediate representation into a formal specification language (i.e., Promela) in order to be able to model check the router configuration.

3.2 Promela Specification

The Promela specification that is derived from an intermediate representation is composed of a set of processes that can communicate with each other. Each process corresponds to a component and it formally describes how a packet is dealt with and how it is forwarded to the next process; there is a one to one

relationship between the intermediate component types and Promela process types (*process source*, *process selector*, *process executor*, and *process sink*).

Promela processes communicate with each other through a set of global variables, that describe the packet properties. These global variables are accessible by all processes that can read and modify them. We have introduced some extra variables for modelling the packet forwarding and the environment; one of them is the variable `exec` that describes the functions that are applied to packets; by monitoring the value of this variable, we can check which functions are applied and in which order.

All processes have a similar structure; a process waits until it receives a packet, then executes its statement, such as routing packets to different processes, or applying functions to them (i.e., dropping). For instance, Figure 4 shows a simplified version of the Promela code for a *selector process* that corresponds to a selector component; the selector process chooses the process to handle the incoming packet; through a sequence of `if` statements, the process checks the value of the packet fields and identifies the next process (setting the variable `next_proc`). The `else` statement is executed if the packet fields do not match any of the other conditions (line 6). We assume that at most one condition is true for a particular value of the packet fields, avoiding a non-deterministic evolution of the process.

```
1. proctype p_2() {
2.  (next_proc==p_id_2);
3.  if
4.  ::(TCPS==1000 && TCPD==1000)->next_proc=p_id_3;
5.  ::(TCPS==1200) -> next_proc=p_id_3;
6.  ::else -> next_proc=p_id_4;
7.  fi;}
```

Fig. 4. Selector Process

A subset of rules defined in Table 1 are mapped into this selector process. For instance, the rule on line 4 corresponds to the rule on line 3 in Table 1, where the rule describes packets from TCP source port 1000 going to TCP destination port 1000.

The router model, which is translated into the Promela specification, handles one packet at a time. This simplification of the model does not limit the analysis of a real router, as we can assume that the presence of a packet does not influence the management of other packets inside the physical router; in fact, the concurrent management of packets speeds up only the router throughput without any influence in the packet handling.

4 Model Checking Promile

Given the Promela specification we can now prove properties on the router configuration using the Spin model checker. The router model must be checked exhaustively to guarantee that the router configuration is acceptable. This implies checking that all packets reaching the router are handled correctly.

To attain this goal, we cannot generate the set of all possible packets, with all possible header field settings, as this would lead to exponential growth of the state space; even considering only the field values referenced by the rules, the packet growth is still exponential. In fact, if we call $\mathbb{P}$ the set of the packets that we need to inject into the router model, and we consider the worst-case scenario where all the rules refer to different field values, the cardinality of this set is shown in (1), where r is the average number of the rules in each module, m the number of modules, and f the number of fields in each rule.

$$|\mathbb{P}| = (m * r)^f \tag{1}$$

$$|\mathbb{P}| <= \sum_{i=1}^{k} \binom{m}{i} * r^i + 1 \ , \ k = min(maxpath, f) \tag{2}$$

However, the cardinality of $\mathbb{P}$ can be limited considering groups of packets, which we will call *packet flows*, instead of single packets, observing the structure of the router configuration and exploiting the relationships among the rules. Following this idea, we can generate less packet flows and reduce the cardinality of the set $\mathbb{P}$ as shown in (2). The cardinality of $\mathbb{P}$ still grows exponentially, but the base is reduced from $m * r$ to r; the exponent is also reduced and is the minimum between the number of fields and the maximum number of the router modules that a packet can traverse ($maxpath$). Moreover, the cardinality of $\mathbb{P}$ is described by an upper bound, reached only in the worst-case. In Appendix A a proof of the fact that (2) is an improvement of (1) can be found. In the next section we give an intuitive justification of the formula in (2).

4.1 Flow generation

In this section we show how reasoning on packet flows instead of single packets allows us to limit the exponential growth of our model. In Section 5, we will then show that flows sufficiently abstract the state space to enable model checks of real-life router configurations.

As packets are identified by a set of fields, a flow can be described by a subset of fields. Following this approach, we group together packets that have some of the fields set to the same value, ignoring the other fields that can assume *potentially* all the possible values. For instance, if the rules inside the router use only two fields (the TCP ports of source (TCPS) and destination (TCPD)), a flow can be described as in (3). Flow f_1 contains all the packets, with fields set to specific values of TCPS and TCPD. In this case, the flow is said to be *fully-defined*, as all fields have a value. On the other hand, flow f_2 is *undefined*

as the value of TCPD is not set; we say that flow f_1 is *more defined* than f_2, as it is a subset of f_2 (5).

$$\mathbf{f_1} = \{(TCPS, TCPD) | TCPS = 1 \wedge TCPD = 1\} \tag{3}$$

$$\mathbf{f_2} = \{(TCPS, TCPD) | TCPS = 1 \wedge TCPD = null\} \tag{4}$$

$$\mathbf{f_1} \subset \mathbf{f_2} \tag{5}$$

In order to prove that the router behaves correctly, we check the router configuration against a set of flows that cover all different paths inside the router. We need to simulate the generation of enough flows so that all the rules inside the modules are matched by at least one flow. By injecting all these flows into the router, the model checker can validate the configuration and pinpoint any router undesired behaviours.

Flows are generated using a Promela specification derived from the intermediate representation of the router. Each component of the intermediate representation is translated into Promela process and, in order to distinguish these processes from the ones that describe the router configuration, we call processes that describe flow generation *network processes*, while *router processes* those that describe the physical router. The connections among the network processes are the same as in the router processes.

The source component is translated into a network process that generates a fully-undefined flow that represents the packets that can reach the router. This flow will be defined more accurately while traversing the selector network processes; the selector network processes correspond to selector components and redefine the packet flow according to their rules. We say that a flow matches a rule when all fields have a compatible value; a flow field is compatible with a rule field if it has the same value or if it is undefined. When a packet flow matches a rule, the selector network process redefines the setting of fields to the value described in the rule. As a flow can match more than one rule, the selector network process randomly chooses one of them.

A simplified network process of a selector component is shown in Figure 5. At line 4 the rule has two fields (TCP source and destination port of the packet); in order to match this rule, the flow has to have the fields undefined or set to same values of the rule.

The sink component is mapped into a network process that behaves as a bridge between the network and the router processes; when it receives a flow, it simply forwards it to the router processes. An executor component is mapped into a forwarding process that receives a flow and forwards it to the next process; the functions applied by this component are irrelevant for the flow generation and they are not modelled.

The number of different flows determines the state space of model checker. The formula in (2) describes the upper bound of the flow set cardinality for the following reason. Consider a simple scenario where packets have only one relevant field (A), and where each module has the same number of rules r. The network source process defines a flow where the field A is set to *null* (i.e., a fully-undefined flow). This flow then traverses the other modules and arrives at the

```
1.proctype np_2() {
2. (next_proc==np_id_2);
3. if
4. ::((TCPS==1000||TCPS==null)
     &&(TCPD==1000||TCPD==null))->
          TCPS=1000;TCPD=1000;next_proc=np_id_3;
5. ::(TCPS==1200||TCPS==null)->
          TCPS=1200;next_proc=np_id_3;
6. ::else -> next_proc=np_id_4;
7. fi;}
```

Fig. 5. Selector Network Process

network sink process. We can argue that a *fully-defined* flow (i.e., where all fields have values) can evolve only through deterministic steps, as there is at most one rule that can match it; on the contrary, a flow where the field is undefined can match all the r rules and then can evolve in r different ways; moreover, when a flow matches a rule it becomes fully-defined, as the field is being set by the rule (and there is only one field per packet in the example). As a consequence, if a non-defined flow reaches every process that has r rules, we obtain (6), where m is the number of modules in our model.

$$\mathbb{P} <= r * m \tag{6}$$

If we want to fully validate the router configuration, we need to also generate a flow matching no rules, so in reality we need to generate $r * m + 1$ flows. This corresponds to the formula in (2) with $f = 1$.

The scenario with $f = 1$ can be extended for a generic value of f. A flow that has f fields can match f different rules in the worst-case, i.e., each rule sets the value of only one field of the flow. For instance, let us consider a flow that is described by two fields, A and B, and a router configuration composed of two modules that have r rules each; moreover, we also assume that the rules inside the first module refer to field A, while the ones in the second module to field B. We now inject a fully-undefined flow; as field A has no value, the flow can match any rule and its field may be set to the value described in one of the rules. The evolution of the flow is non-deterministic and can evolve in $r + 1$ different ways (matching r different rules or matching none). When this set of $r + 1$ flows reaches the second module, the flow evolution is more complex. Each flow can match any of the rules, as field B is undefined. Moreover, the flow can traverse the module without matching any rule. Therefore, the $r + 1$ flows can match all the rules, and we obtain $(r + 1) * r$ flows, or they do not match any rule and we are left with $r + 1$ flows. Then we can deduce the formula:

$$(r + 1) * r + r + 1 = r^2 + 2 * r + 1 = \binom{2}{2} * r^2 + \binom{2}{1} * r + 1$$

This formula is formula (2) with $f = 2$. The idea can be extended considering m modules and f fields obtaining formula (2) .

4.2 Proving Router Properties

We can prove that the router configuration complies with a set of defined properties. The property validation is based on a simplification of the router model; the router model is able to manage only one packet flow at a time, and moreover, after that, it is turned off. Obviously this behaviour is not the same of a physical router, but, if we can prove that the router model is able to handle correctly a single packet flow, then the real router will be also able to handle a sequence of packets.

The prove of some properties is embedded into the specification; for instance, we can deduce which rules are not used by reasoning about the unreachable code of the Promela specification, and find out which rules or modules should be removed from the router configuration. Moreover, through the *assertion* mechanism supported by Spin and embedded in the specification, we can verify that no packet is going to loop inside the router simply asserting that no module will receives two, or more, packet flows.

Finally, we can define generic properties using Linear Temporal Logic (LTL). Through LTL, we can define the sequences of functions that have to be applied to every packet and forbidden sequences. In this way, we can formalize the *router*, *service*, and *performance properties*, as defined in Section 2.5. For instance, a *service* property is described in (7). This property requires that the packet that belongs to a particular host will never be dropped; the packets are identified through the term $fromHost(A)$. The other LTL property (8) is concerned with router performance. This property forbids dropping a packet that was previously marked inside the same router.

$$\Box(fromHost(A) \implies \Box(\neg dropPacket())) \tag{7}$$

$$\Box(markPacket() \implies \Box(\neg dropPacket())) \tag{8}$$

5 Evaluation

Our work aims at minimizing the validation time of a router configuration. This operation must be flexible enough to accommodate any network change and must be relatively fast to be used interactively by the administrator. According to the requirements of the network administrator, the validation can be done on-line before applying the operation, or off-line if the router update cannot be delayed (e.g. during a *denial of service attack*).

We have tested several realistic router configurations and in particular we have evaluated two different configurations: a 1-level tree and a sequence of modules. We have chosen these two configurations so that our approach reaches the best and the worse performance, respectively. In fact, the number of generated packet flows is related to the number of modules that a packet can traverse; in the first case, there are only 2, the root and the leaf, while in the second case all the modules. The two configurations have the same number of modules (seven),

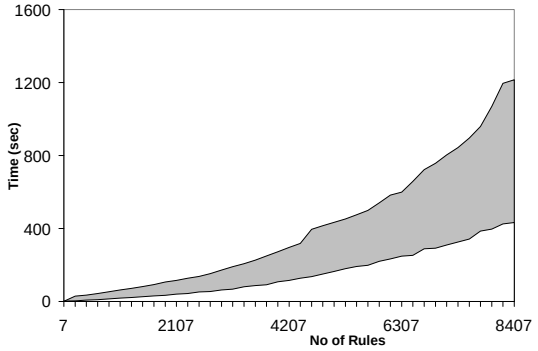

Fig. 6. Model Checker Performance

and the rules concern the same fields (three), while the number of rules goes from 7 to 8,500. We have chosen to change the number of rules, as, from our experience, the number of modules or fields is usually three or four orders of magnitude lower than the number of rules; it is then more relevant to prove how our approach scales according to rules, instead of other parameters. As these two configurations produce the worse and the best performance of our tool, they also represent an upper and a lower performance bound for a router configuration with the same number of modules, rules, and fields.

In Figure 6, the lower and the upper performance bounds are enclosed in the gray area. Interpolating the result, we have obtained the two equations shown in (9) and in (10). Both of them have a small coefficient of the second order and in particular (9) can be approximated to a line.

$$y = 0.2609 * x^2 - 0.3319 * x + 8.4093 \tag{9}$$

$$y = 0.7653 * x^2 - 4,6192 * x + 56,287 \tag{10}$$

These tests have been performed on a PC equipped with a Xeon@1.7GHz processor and with 1 GB of RAM. Note that a configuration with seven modules and a total of 8,500 rules is a rather complex one for this specific application domain. We consider it remarkable that on a relatively small off-the-shelf PC we are able to validate the worst case configuration in less than 20 minutes.

6 Related Work

The related work to this project belong from different research areas. From the networking research area, there are two interesting projects: Click [9] and Router Plugins [5]. Click and Router Plugins are implementations of programmable routers, where the packet paths inside the router (i.e, the modules and their connections) can be configured; the abstraction provided by Click is more complex than our kernel, as there are more port types in order to describe the

behaviour of the module more precisely. Furthermore, Click and Router Plugins do not provide a middleware layer to support the network administrator during the configuration process. However, we believe that the principles outlined in this paper could also be applied to these projects. From the software engineering perspective, model checking techniques, as data and model abstraction, are been investigated in [3]. The Bandera toolset [4] can verify that generic Java code complies with a set of properties defined by users; this is achieved through model slicing and abstraction; in particular, the abstraction is manually driven by users through a graphic interface. In our approach, the abstraction of the model is provided by the middleware, that shows to the model checker tool a graph of modules; as the domain of our tool is always the same, we have implemented an automatic abstraction of the input data (i.e., the packets) that is completely transparent for the network administrator and provides better performance than a general purpose one; however this approach is domain specific and cannot be applied to generic scenarios.

Model checking techniques are also applied to networking protocol research [7]. In this domain, the system is distributed and it is relevant to cope with its embedded concurrency; through model checking as shown in [6], the network protocol is specified using Promela and validated through Spin. The advantages of this approach are mainly two. Firstly, the network protocol is designed using a specification language (i.e. Promela). Secondly, the specification can be validated using a model checker (i.e., Spin). Spin is used to check the correctness of the network protocol, but it does not cope with the network configuration. In fact, it assumes that the network topology is correct and that it should work correctly also if there are some errors. Spin checks the robustness of the protocol, but it does not check the network environment. The aim of our work is complementary: we do not manage the router life-cycle, but we model check its configuration in order to guarantee that it will work correctly and that it will comply with the required properties.

7 Conclusions and Future Work

In this paper we have described an approach to management of programmable networks that takes advantage of model checking techniques in order to prove that router configurations are consistent and safe. The model checker is integrated into a tool that allows the network administrator to manage a network, confidently update configurations at run time. The tool is based on a formal description that can be translated into Promela, the specification language of Spin. Exploiting the model checker Spin, we can prove that the router configuration is correct or, report errors to the network administrator. In terms of analysis, we are currently able to prove that a router is providing the right services in the right way, but we cannot check that the whole network is working as required.

The work presented in this paper shows how application-specific languages can be designed by applying software engineering principles, how their semantics

can be defined and most importantly that it becomes possible to apply model checking to real-life problems by exploiting domain-specific abstractions.

The next steps of this work will go in two different directions; through the tool, the network administrator will be able to *draw* the router properties instead of writing LTL formulas; moreover, the extension of the tool domain is relevant in order to move the verification from a single router to a network of routers.

Acknowledgments: We would like to thank Karen Page for her assistance in the proof. We thank the `Promile` group and Licia Capra for her comments on an earlier draft. We acknowledge the financial support of EPSRC, BT, and Kodak.

References

1. S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. IETF-RFC 2475 - An Architecture for Differentiated Services, December 1998.
2. A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communications Review*, 29(2):7–23, Apr 1999.
3. E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448, Limerick, Ireland, June 2000.
5. D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Route plugins: A software architecture for next-generation routers. *IEEE/ACM transactions on Networking*, 8(1):2–15, February 2000.
6. E. Gauthier, J.-Y. L. Boudec, and P. Oechslin. SMART: A many-to-many multicast protocol for ATM. *IEEE Journal of Selected Areas in Communications*, 15(3):458–472, 1997.
7. P. Holzman. *Design and Validation of Network Protocols.* Prentice Hall, 1991.
8. G. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
9. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
10. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schaefer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, pages 137–153. Springer-Verlag, Berlin, 1995.
11. J. Magee and J. Kramer. *Concurrency: Models and Programs – From Finite State Models to Java Programs.* John Wiley, 1999.
12. H. D. Meer, W. Emmerich, C. Mascolo, N. Pezzi, M. Rio, and L. Zanolin. Middleware and Management Support for Programmable QoS-Network Architectures. In *Short Papers Session of the 3rd Int. Working Conference on Active Networks (IWAN)*, Philadelphia, PA, October 2001.
13. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science(FOCS-77)*, pages 46–57, Providence, Rhode Island, October 1977.

A Proof

Hypothesis:

$$\sum_{i=1}^{k} \binom{m}{i} * r^i + 1 < (m * r)^f \tag{11}$$

We approximate assuming the difference between the two terms is bigger than 1:

$$\sum_{i=1}^{k} \binom{m}{i} * r^i < (m * r)^f \tag{12}$$

With:

$$\begin{aligned} m, r, f, k > 0, m \geq k, f \geq k \\ k = min\{f, maxpath\} \end{aligned} \tag{13}$$

$$\sum_{i=1}^{k} \binom{m}{i} * r^i < \sum_{i=1}^{k} \binom{m}{i} * r^f < (m)^f * r^f \tag{14}$$

$$\sum_{i=1}^{k} \binom{m}{i} < (m)^f \tag{15}$$

The first term of (15) is indeed less than m^f because:

$$\sum_{i=1}^{k} \binom{m}{i} \leq \sum_{i=1}^{f} \binom{m}{i} = m + \sum_{i=2}^{f} \binom{m}{i} \tag{16}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} \sum_{i=2}^{f} \frac{1}{i!} \tag{17}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} (\frac{1}{2} + \frac{1}{2*3} ... + \frac{1}{2*3*...*f}) \tag{18}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} (\frac{1}{2} + \frac{1}{2*2} ... + \frac{1}{2*2*...*2}) \tag{19}$$

As the last term in (19) is less than 1 we can approximate:

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} \tag{20}$$

and for f larger than 2:

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m^f \tag{21}$$