Model Checking Programmable Router Configurations

Luca Zanolin, Cecilia Mascolo and Wolfgang Emmerich
Dept. of Computer Science
University College London
Gower Street
London WC1E 6BT
{L.Zanolin|C.Mascolo|W.Emmerich}@cs.ucl.ac.uk

ABSTRACT

Programmable networks offer the ability to customize router behaviour at run time, thus providing new levels of flexibility for network administrators. We have developed a programmable network framework and used it to implement Differentiated Services (DiffServ) in order to guarantee network quality of service. Part of the framework is an application-specific language that we use to program router configurations. The ability to change these configurations dynamically introduces the danger of router misbehaviour. We show an approach that uses model checking techniques to analyze router configurations prior to their deployment in a router. Our approach allows network administrators to specify correctness criteria and check whether or not specific router configurations meet these criteria.

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 they forward packets to another router that is closer to their 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, classes of applications, such as audio-conferencing or download of streamed video that would be improved if quality of service (QoS) guarantees could be provided by the network. Moreover, there are different classes of users in the Internet. A company might be prepared to pay a premium if they are provided with 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 a number of interesting software engineering research questions. Firstly, it is desirable to provide a high-level and application specific programming language to allow network administrators to write router programs at appropriate levels of abstractions. At the same time, this language must be efficiently executed in order to avoid compromising network and routing performance. Secondly, router programs need to be changed on a regular basis in order to introduce new services without having to shut down the routers. Thirdly, the network typically contains a large number of routers that might be quite heterogeneous and we would like to hide this heterogeneity from network administrators. Finally, prior to updating a single router or a set of routers with a new router program, we would like to enable network administrators to check that their programs do not compromise the overall network performance and reliability.

The main contribution of this paper is a solution to the last 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 [7]. 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. Focusing on the software engineering aspect of this research, 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 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 present the architecture and the implementation of our tool and we evaluate it in Section 6. Section 7 compares our approach to related work and in Section 8 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.

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. We may, for example, assign a class to packets that belong to a video conference application and another class to packets that belong to a generic download session. For the first class we have to guarantee that the packet flow is uniform and the application would not benefit from speeding up the transmission, even when it is allowed. On the other hand, when users are downloading data, they are interested in maximizing network throughput. In this scenario, the video conference packets should have priority only if the network is congested, otherwise the download packets should be prioritized.

We have implemented a programmable router, called `Promile` [12] and among other applications, we have programmed it for DiffServ. As shown in Figure 1, `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 and the services. `Promile` has two main advantages: firstly, we can change the router configuration at run-time without traffic disruption. Secondly, we can manage the router through a middleware that hides all the implementation details, allowing the network administrator to work at a higher level of abstraction.

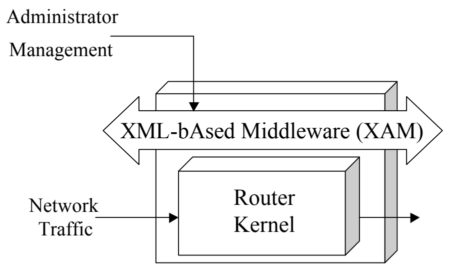

Figure 1: `Promile` two layer architecture

XAM presents the administrators with an abstraction of the routing machine, allowing them to modify the router to provide new services or modify old ones. We call the router abstraction *configuration*, because it configures how the router manages packets. The configuration is a high level abstraction and it is 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.

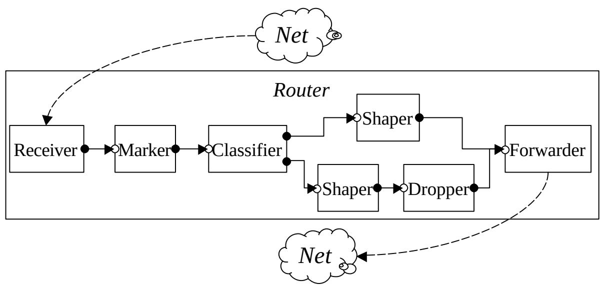

Figure 2: Router Configuration

Figure 2 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 the speed of the transmission, 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 class; for instance, the video conference service corresponds to the first class, while the download service to the second one.

As shown in Figure 2, the *Receiver* module receives packets from the network and just 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.

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.

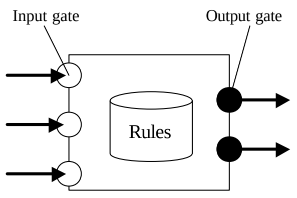

Figure 3: A `Promile` Module

A module (Figure 3) 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. For instance, packets received from a particular input gate may belong to a particular class of service and the module may delay them without checking their headers. Symmetrically, we can have more than one output gate if we need to send different packets to different modules. For instance, if a Marker assigns a class to two incoming packets, they may be sent through different output gates. In this way, the router avoids a performance penalty as other modules may not have to check packet classes. 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 is the one that re-injects the received packets into the network; this kind of module is called *sink module* and it models an output network interface. When a packet reaches the router, it is firstly managed by a source module and, after traveling through a configuration, it is sent to the next hop by a sink module.

`Promile` distinguishes the notion of *module types* and *module instances*. A router program may instantiate multiple modules of the same type and configure them in a different way. This provides flexibility in the definition of the service, as we can introduce new kinds of services independently from other module instances on the same the router. Moreover, when network administrators want to modify the router configuration, they can refer to the modules also by their types and, for instance, decide to replace all the module instances of a particular type with a newer version.

2.2 Connections

Connections describe the packet flow between modules. Connected modules form a directed graph like the one in Figure 2. A connection starts from an output gate and leads to an input gate. Any output gate must be connected to an input gate to specify the packet forwarding from module to module; moreover, only one connection can start from an output gate, otherwise the packet forwarding would not be deterministic. Multiple connections may end in the same input gate and they can also describe a cyclic graph among the modules.

2.3 Rules

We have described modules and their interconnections using gates. We now illustrate how module instances can be configured through rules. The functions of a module are not applied to all incoming packets, but only to a sub-set of these. These subsets are identified by a set of rules.

A *rule* describes a packet through a set of fields related to the packet header. A rule can, for instance, identify a packet flow that originates at a particular host, so that a specific function, such as *drop* is applied to packets coming from that host. The set of rules that configures the module behaviour is flexible and can be updated in order to modify the set of packets on which the module function should be applied. The fields inside a rule can change according to the module whose behaviour it controls.

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 an example rule set. These rules control the behaviour of a *Marker* module and they describe how classes are assigned based on source and destination IP addresses. The rules are composed of four fields about the packet header plus one that describes how to mark the packets (i.e., class). The first rule identifies the packet class by specifying both its source and its destination address and assigns it to the *first class*. The second, third and fourth rules specify only a sub-set of the fields. The third rule, for instance, determines that packets that are sent from TCP port

1000 to any destination on TCP port 1000 belong to class 10. Using these rules, we can flexibly parameterize the behaviour of individual module instances.

2.4 Configuration Update

The module instances, their interconnections, as well as the rules that have been used to parameterize 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. For instance, the network can react to a *denial of service attack* deciding to accept, for a short time, only packets coming from a trusted set of hosts.

When the network administrator, or an automatic tool, wants to update the router configuration, they only have to deal with the router abstraction provided by `Promile`. The abstraction is useful as it simplifies the network management and hides the implementation details of routers. The router configuration can be updated by applying a sequence of actions; for instance, an action can insert new module instances, update the rules of a module instance or establish new connections between module instances.

Prior to defining a new configuration, the administrators want to convince themselves that the new configuration is meaningful. In order to do so, they should be enabled to specify invariant properties that they would like to assert on all consistent configurations.

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 packets should be managed independently from services. For instance, a routing property could describe that a packet cannot loop inside the router. In general, a routing property guarantees that the router always works correctly and that it is always able to handle new incoming packets. As the routing properties are not related to the service provided by the router, they are almost the same for all the routers in the network and they are generally defined by the network administrator. Property (1) shows a routing property in linear temporal logic (LTL); the term $receivePacket()$ is true when the router receives a packet, while the term $sendPacket()$ and $dropPacket()$ are true, respectively, when the packet is sent or dropped. The routing property below describes that all packets received by the router must be thereafter explicitly sent or dropped; as a consequence, it requires that no packets may accidentally be lost inside the router.

$$\Box(receivePacket() \implies \Diamond(sendPacket() \vee dropPacket())) \quad (1)$$

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 or by module developers. For instance, an administrator can require that some functions are never applied to particular packets. Property (2) requires that a particular packet flow is never dropped; the flow is identified by the term $fromHost(A)$ that is true when the packets are sent from host A.

$$\Box(fromHost(A) \implies \Box(\neg dropPacket())) \quad (2)$$

Performance properties are concerned with the router performance and they try to maximize router throughput. 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 developer of the router. For instance, we may want to forbid that a sequence of functions is applied to a packet; if a packet is marked and then dropped, the router wastes time marking the packet that, instead, should have been dropped straight away. This last property is formalized as (3). The property states that if a packet is marked, it cannot be dropped by the same router.

$$\Box(markPacket() \implies \Box\neg dropPacket()) \quad (3)$$

Another performance property that should be checked is concerned with rules and modules. According to the router packet flow, some rules could be irrelevant, because any packet matches them. In order to increase the router speed they should be removed. We should also check if all the modules are reachable by at most one packet, otherwise we should remove them.

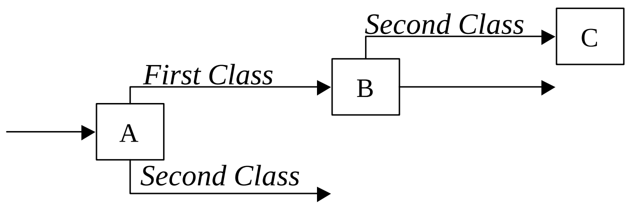

Figure 4: Not-reachable rule

For instance, as shown in Figure 4, module A has a rule that describes that all first class packets must be forwarded to module B; similarly, there is a rule in module B that forces all *second class* packets to be processed by module C. According to the configuration shown in Figure 4, we can argue that no packet can match the rule in module B and it should be removed. Moreover, module C can only receive packets from module B, but as we already said, this cannot happen; as a consequence, module C is not reachable and should be removed.

3. OPERATIONAL SEMANTICS

In order to prove that the router satisfies desired properties, we need to translate the router configuration into a formal specification language. We can then apply model checking techniques to validate the router configuration against the desired properties. In doing so, we also give an operational semantics to the router configuration language that we have so far only described informally.

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 one, starting from this representation, produces a Promela specification that defines the operational semantics and can be model checked by Spin. Splitting this translation into two steps has two advantages. 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. The second advantage is that the intermediate representation provides a comprehensible documentation of a router configuration for network administrators. We now describe the intermediate representation and its mapping to the Promela language.

3.1 Intermediate representation

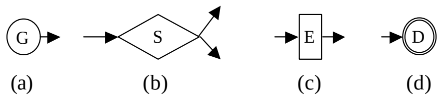

Figure 5: Components

The intermediate representation supports four component types: *generator* (G), *selector* (S), *executor* (E), and *destroyer* (D) (Figure 5). A *generator* (Figure 5a) 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 5b) 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; all the rules inside the selector have the same format and they refer to the packet header and to environment variables. A selector has one input and a list of numbered output; if a received packet matches no rules, it is sent through the default output (0). An *executor* (Figure 5c) receives packets and forwards them after applying a function. An executor applies the same function to all packets that it processes. A *destroyer* (Figure 5d) receives packets and destroys them; arrows denote connections between components and they determine packet flows.

Third party module developers have to provide translations between their module types and this intermediate representation in order to provide semantics for their modules. For instance, a DiffServ module of type *Dropper* could be mapped into the intermediate representation shown in Figure 6. The semantics of this module type is described using a selector component (S) that chooses the packets that should be dropped and forwards them to the executor component (E) that applies the module specific function (i.e., drops packets) and sends them to the destroyer component (D), which destroys them[1].

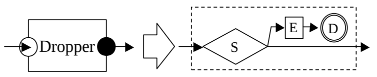

Figure 6: Translation of a Dropper Module into Intermediate Representation

Figure 7 shows how the router configuration in Figure 2 is mapped using the four component types that we have introduced. The rules defined in Table 1 need to be incorporated into the model as they define the specific behaviour of the different modules. This operation is trivially done importing them into the *selector* component of each module. The *selector* is the only component containing rules.

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.

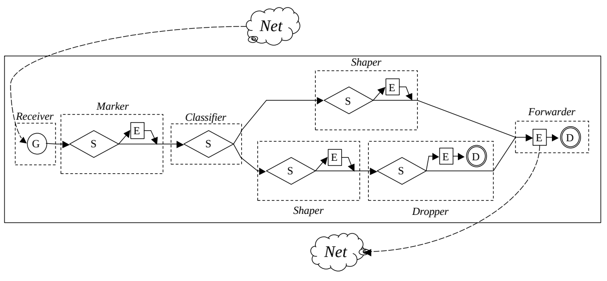

Figure 7: Component Graph

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

[1]In this particular case the packets are virtually dropped in the executor component and physically destroyed in the destroyer component.

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 generator*, *process selector*, *process executor*, and *process destroyer*).

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 also introduced two additional global variables in order to describe the evolution of the packet management inside the router; an important one, called `next_proc`, models the connections between components and describes which is the next process that has to handle the packet, and that we use to map the intermediate graph representation into Promela processes interactions.

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 8 shows 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;}
```

Figure 8: Selector Process

The rules defined in Table 1 are mapped into the selector process. An example of rule is shown on line 4. This rule corresponds to the rule on line 3 in Table 1.

4. MODEL CHECKING PROMILE

Given the Promela specification we can now prove properties on the router configuration using the Spin model checker.

4.1 Abstractions to Reduce the State Space

A router configuration must be checked exhaustively to have the guarantee that the router works correctly. This implies checking that all the packets reaching the router are handled correctly. To attain this goal, we generate all the possible packets, with all the possible header field settings and use them to check the model. We can then verify that all paths followed by the packets are acceptable. However, the cardinality of the set $\mathbb{P}$ of possible packets is significant and influences the time that the model checking tool takes to validate the router configuration.

For instance, if the rules used in a configuration have four fields (i.e., TCP port and IP address of source and destination, like in Table 1), we need to use 12 *bytes* (2 bytes for a TCP ports and 4 bytes for an IP address) in order to store them and to generate 256^{12} different packets. Thus the cardinality of $\mathbb{P}$ grows exponentially with the number of fields.

In order to improve the performance of the model checker we can see that the number of packets we need is limited to those that are referenced in the modules. This can be written as (4), where r is the average number of rules in each module, f the number of fields in each rule, and m the number of modules. Formula (4) is still exponential in the number of fields.

$$|\mathbb{P}| = (m * r)^f \tag{4}$$

$$|\mathbb{P}| <= \sum_{i=1}^{k} \binom{m}{i} * r^i + 1 \tag{5}$$
$$k = min(maxpath, f)$$

However, the cardinality of $\mathbb{P}$ can be limited considering groups of packets (i.e., flows) instead of single packets, observing the structure of the router configuration and exploiting the relationships among the rules. The formula we obtain is shown in (5). 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 worse situation. In Appendix A a proof of the fact that (5) is an improvement with respect to (4) can be found. In the next section we give an intuitive justification of the formula in (5).

4.2 Flow generation

In this section we show how reasoning on packet flows instead of packets allows us to limit the exponential growth of our model.

As packets are identified by a set of fields, a flow can be described by a sub-set 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 (6). 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 (8).

$$\mathbf{f_1} = \{(TCPS, TCPD) | TCPS = 1 \wedge TCPD = 1\} \tag{6}$$
$$\mathbf{f_2} = \{(TCPS, TCPD) | TCPS = 1 \wedge TCPD = null\} \tag{7}$$
$$\mathbf{f_1} \subset \mathbf{f_2} \tag{8}$$

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. After injecting all these flows into the router, the model checker can validate the configuration and point out any router undesired behaviours.

Flows are generated using the model checker itself and by modeling the network through a set of Promela processes. In Section 3 the network is implicitly described as a packet generator and is not modelled. In order to efficiently prove that a router configuration satisfies the desired properties, we need to identify the packets that reach the router and devise a network model. The model should only generate the minimal set of packet flows needed to validate the router configuration, in order to reduce the state explosion of the model checker. The network model is similar to the one used for the router (Figure 7) and, again, it contains generator, selector, executor and destroyer processes. Each module corresponds to a combination of processes [2]. In order to distinguish these processes, we call the ones that belong to the router *router processes* and the ones to the network *network processes*. The network processes describe the same structure of the router configuration, but, differently from router processes, they can evolve in a non-deterministic way. At the beginning we inject a fully-undefined flow (i.e., all the fields are set to `null`) into the network model. The processes in the model progressively set fields of the packets to cover all possible paths inside the router. At the end, the network model produces, through a non-deterministic evolution, a single flow; after an exhaustive validation, the network model generates all the flows needed to check the router configuration exhaustively. These flows are then injected into the router model shown in Figure 7.

The number of different generated flows determines the time taken by the model-checker to prove the correctness of the configuration of the router. The formula in (5) describes the upper bound of the flow set cardinality and we now explain its meaning. For simplicity, we consider a simple scenario where packets have only one relevant field (TCPS), and where each module has the same number of rules r. The network generator process defines a flow where the TCPS field is set to $null$ (i.e., a fully-undefined flow). This flow then traverses the other modules and arrives at the network destroyer process: how many different evolutions of the modules are allowed? 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 into r different way; 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 (9), where m is the number of modules in our model.

$$\mathbb{P} <= r * m \tag{9}$$

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 (5) with $f = 1$.

We have considered the scenario with $f = 1$. Now we describe how it can be extend for a generic value of f. A flow that has f fields can match f different rules, in the worse case, when 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 assume that the rules inside the first module refer to field A, while the ones from 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 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 (5) with $f = 2$. The idea can be extended considering m module and f fields obtaining formula (5) .

4.3 Proving Router Properties

After a flow has been generated by the network processes, it is forwarded to the router processes that model the router configuration. We can define the model properties using Linear Temporal Logic (LTL) and we can check that all the specification code is reachable. Through LTL, we can define the sequences of functions that have to be applied to every packets and forbidden sequences. We can formalize both *router* and the *service properties*, as we have defined them in Section 2.5. A property saying that a packet with a specific TCP port destination should not be dropped, is described in (10) and it is a refinement of the property sketched in (2). The variable `exec` used to specify these properties is one of the global variables defined in the Spin specification together with `next_proc` (Section 3). It is used to hold the name of the function that is applied to packets.

$$\Box(exec = start \implies \Box(TCPD = 10 \implies exec \neq drop)) \tag{10}$$

[2] A selector process containing the rules is present in each module, as described in Section 3.

Performance properties of the router configuration do not affect the functions applied to the packets but concern performance; in other words, these properties pinpoint configurations that slow down the router. For example, redundant rules should be removed as they slow down a module while it is determining if a packet matches those rule. A rule is defined as *redundant* if there is no packet that matches it. Through model checking, we can easily identify redundant rules; in fact, Spin can isolate unreachable lines of the specification, and the relationships between these lines so that those rules can be automatically determined.

One very important property is that packets should not be allowed to loop forever in the router. This property is proved through an LTL formula and can be proved against the router configuration in order to obtain the wanted guarantee. Through (11), we require that all the packets reach the destroyer process, meaning that they are not looping forever in the router.

$$\Box(exec = null \implies \Diamond(exec = start \implies \Diamond\Box exec = end)) \quad (11)$$

5. IMPLEMENTATION

In Section 2 we have introduced the architecture of `Promile`, that is composed of two layers: an Xml-bAsed Middleware (XAM) and the Router Kernel. The Router Kernel manages the packet forwarding process, while XAM is the middleware that manages updates to the configuration of the router. The requirements of these two layers are different as they have different goals. Briefly, XAM must be portable and flexible to run on different hardware platforms, while the kernel must be fast to maximize the router throughput. According to these requirements, we have implemented XAM in Java and the kernel in C. The two layers communicate through Java Native Interface (JNI) [9]. XAM is designed to work on different Programmable Routers. Our Router Kernel, in fact, is only one of the possible implementations on which XAM can run.

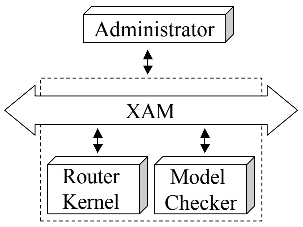

Figure 9: Promile Architecture

As shown in Figure 9, XAM interfaces the network administrator with the router kernel. The model checker is a separate component and is supposed to run on a different processor, for performance reasons. The administrator can use a visual tool (Figure 10) to set up a new configuration that is then model checked prior to being sent to the kernel. These steps are completely transparent to the network administrator, who only deals with the graphical router configuration language.

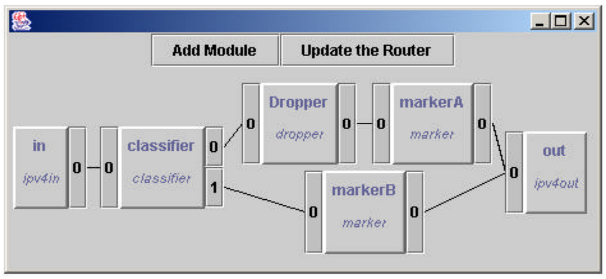

Figure 10: Administrator Visual Tool

XAM stores the router configuration as an XML document that can be manipulated using open source libraries that implement the WWW consortium's Document Object Model (DOM) and XPath. Through those tools, XAM is able to manage the router configuration as a tree and a module is seen as a node. Using DOM, XAM can manipulate the tree and using XPath it can perform complex queries on the tree (i.e., find a module connected to another module and so on). A first syntactic check of the configuration is made by XAM by validating a router configuration against its XML Schema. Through XML Schema, we define the grammar of the router configuration language and we can check that a new configuration is syntactically correct. After that, XAM has also to prove that the new configuration complies with a set of defined properties and, in order to achieve that, it has to transform the configuration into the intermediated representation and then into the Promela specification that can be analysed by the model checker. The router configuration is stored as an XML document and through a style sheet transformation (XSLT), XAM translates the configuration into the intermediate representation. We use style sheets to enable third party module designers to provide a semantics mapping of their modules to the intermediate representation (Figure 11).

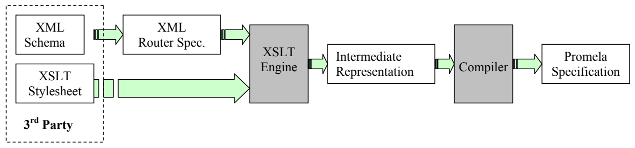

Figure 11: Translation Steps

The second translation step into Promela is implemented in Java for performance reasons. If Spin finds an erroneous configuration, the error is shown to the administrator via the visual tool. A full description of the architecture of XAM and and the Router Kernel can be found in [12].

6. EVALUATION

Our work attempts to minimize the time for model checker to validate a router configuration. On the one hand, the router must be flexible enough to accommodate any possible network change. On the other hand, the network administrator wants to validate the router configuration and, for a real

DiffServ scenario, this operation can take several minutes. These two requirements seem contradictory but if the network administrator has to update the router in order to introduce a new service, it is reasonable to spend some minutes model checking the configuration in order to be sure that it is consistent and safe. On the other hand, if the network administrator has to perform a quick update of the router as a consequence of a *denial of service attack*, the update can be performed immediately and the model checking validation can be deferred.

We have made several tests using realistic router configurations and in particular we have used 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. The two configurations have the same number of modules (seven), and the rules concern the same fields (three), while the number of rules in each modules go from 7 to 8,500. Through these tests, we are able to estimate a lower and upper bound for the time that the model checker takes to validate a router configuration composed of seven modules where the rules affects only three fields.

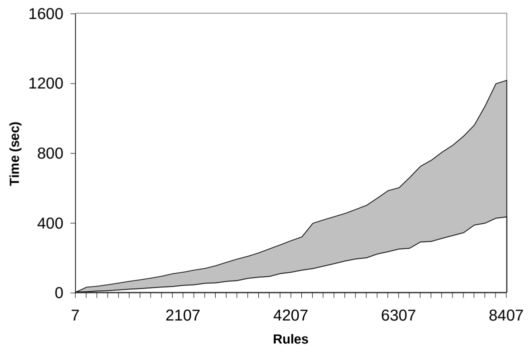

Figure 12: Model Checker Performance

In Figure 12, the lower and the upper bounds are enclosed in the gray area. Interpolating the result, we have obtained the two equations shown in (12) and in (13). Both of them have a coefficient of the second order close to zero and in particular (12) can be approximated to a line.

$$y = 0.2609 * x^2 - 0.3319 * x + 8.4093 \tag{12}$$
$$y = 0.7653 * x^2 - 4,6192 * x + 56,287 \tag{13}$$

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 8000 rules is a rather complex one. 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.

7. RELATED WORK

The related work to this project belong to two different research areas: networking and software engineering. From the networking research area, there are two interesting projects: Click [8] and Router Plugins [3]. Click is an implementation of a programmable router, 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. Nevertheless, all the effort in this project focuses on the networking layer; Click does not provide a middleware layer to help the network administrators. Furthermore, Click does not support analysis of configurations. However, we believe that the principles outlined in this paper could also be applied to Click. The Router Plugins approach [3] is another implementation of programmable routers. As for Click, Router Plugins does not provided any middleware and it does not support any analysis of router configurations.

From the software engineering perspective, model checking has recently gained significant attention. Java Path Finder [5] allows to check that generic Java code complies with a fixed set of properties. The main contribution of Java Path Finder is the ability to model check general concurrent Java code, while we apply model checking to an application-specific programming language.

It is probably fair to say that Spin had its roots and motivation in network protocol research [6]. Model checking techniques are also applied to networking protocol research. In this area, the system is distributed and it is relevant to cope with its embedded concurrency; through model checking as shown in [4], 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.

Our work has been influenced by graphical ADLs, and in particular by Darwin [10], which describes a system configuration through components, ports, and connectors. Like Darwin, we are able to dynamically update a configuration and apply model checking techniques to it. The difference is that our configuration language has been explicitly geared toward its application domain and is directly executed by our programmable router's forwarding engine. Also, Darwin uses reachability analysis of labelled transition systems for model checking [11], while we use Spin, a more powerful model checker that allows us to check any LTL formula.

8. 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 visual 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 through a visual tool.

As the complexity of model checking a router can become quickly unmanageable, we have exploited the relationships between the rules that can be deduced from the router configuration in order to simplify the model. The obtained result from the tests suggests that our approach can be applied in realistic DiffServ networks providing good performance.

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 next step of this work should be to extend our model from a router to a network of routers.

Cisco have recently extended their product portfolio with a programmable router product. Cisco have agreed to provide us with this Intelligent Engine 2100 to which we will port `Promile`. In order to make the basic research results presented here more relevant for use in practice, we are going to investigate how the model checking techniques shown in this paper can be applied to Cisco IOS router configurations.

Acknowledgments: We would like to thank Karen Page for her assistance in the proof shown in the appendix. We thank Saleem Bhatti, Jon Crowcrof, Hermann de Meer, Davide Lamanna, Karen Lawson, Nicola Pezzi, and Miguel Rio for their support on the Promile project, and Licia Capra for her comments on an earlier draft. We acknowledge the financial support of EPSRC and Kodak.

9. 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] 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), July/August 2000.

[4] 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.

[5] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000.

[6] P. Holzman. *Design and Validation of Network Protocols*. Prentice Hall, 1991.

[7] G. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[8] 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.

[9] S. Liang. *The Java$^{(Tm)}$ Native Interface: Programmer's Guide and Specification*. Addison Wesley Longman, Inc., 1999.

[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 3^{rd} Int. Working Conference on Active Networks (IWAN)*, Philadelphia, PA, Oct. 2001.

[13] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

APPENDIX

A. PROOF

Hypothesis:

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

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

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

With:

$$\begin{gathered} m, r, f, k > 0, m \geq k, f \geq k \\ k = min\{f, maxpath\} \end{gathered} \tag{16}$$

$$\sum_{i=1}^{k} \binom{m}{i} * r^i < \sum_{i=1}^{k} \binom{m}{i} * r^f < (m)^f * r^f \tag{17}$$

$$\sum_{i=1}^{k} \binom{m}{i} < (m)^f \tag{18}$$

The first term of (18) is indeed less than m^f because:

$$\sum_{i=1}^{k} \binom{m}{i} \leq \sum_{i=1}^{f} \binom{m}{i} = m + \sum_{i=2}^{f} \binom{m}{i} \tag{19}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} \sum_{i=2}^{f} \frac{1}{i!} \tag{20}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!}(\frac{1}{2} + \frac{1}{2*3} ... + \frac{1}{2*3*...*f}) \tag{21}$$

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!}(\frac{1}{2} + \frac{1}{2*2} ... + \frac{1}{2*2*...*2}) \tag{22}$$

As the last term in (22) is less than 1 we can approximate:

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m + \frac{m!}{(m-f)!} \tag{23}$$

and for f larger than 2:

$$m + \sum_{i=2}^{f} \binom{m}{i} \leq m^f \tag{24}$$