NetSAT

Automated reasoning methods for verification and configuration of computer networks

Marco Gario

Abstract

In this work we explore the possibility of using well known methods from different areas of automated reasoning to verify and correct the configuration of computer networks. After providing a rigorous formalization of the problem we present possible encodings and a working solution based on the Planning as SAT method. Finally we introduce some complexity results and the prototype NetSAT.

Contents

1. Introduction

1.1. Goal overview

The problem we are interested in solving is the following: Given a computer network of which we know the configuration and the intended behaviour (*policy*), we want to check whether the current configuration "satisfies" the policy or not; if not we want to know what are the alternative configurations that can satisfy it (if any).

The complete description of the problem will be composed by:

- Description of the network (Section 2, 3)
- Policy (Section 4)
- Configuration(s) (Section 5)

1.2. Planning, Model Checking and SAT

Solving this problem can be seen both as a planning and as a model checking task. In fact when we think about the reconfiguration of the network, we can see how this fits well as a planning problem. On the other hand, when we think about testing policy satisfaction, we can easily think about specifications in a model checking problem. These fields are strictly related to each other, so it is not surprising that our problem can be reduced to a planning or a model checking problem. Since both problems can be solved by a reduction to SAT ([10], [16]) we decided to deal directly with the SAT problem. This gave us the opportunity to tweak the encoding, when suitable, and have, therefore, a better insight on the problem.

1.3. Related works and contributions

The problem presented in this work is partially covered by other works. In particular two solutions are worth mentioning: configAssure ([20]) and ConfigChecker ([5],[6]).

configAssure configAssure was developed to find a suitable configuration satisfying a set of requirements. It also takes advantage of SAT solving techniques and it is divided into two components: a prolog *Requirement Solver* and the KodKod relational solver ([27]). After pre-processing the constraints through the Prolog engine, configAssure invokes KodKod to find a solution to the constraint problem; in turn the problem is reduced to SAT and a model returned (when possible). The requirement solver allows to deal with problems of bigger size: many trivial solutions or contradictions are solved at this stage using higher level domain knowledge. Unfortunately this work is part of a commercial applications ([25]), and little information is provided on the modelling of the various components and on the complexity of the process.

ConfigChecker ConfigChecker deals only with the verification of a given network according to some requirements. In this context, this solution is more similar to ours, and quite interesting. Even though ConfigChecker uses an extension of computation tree logic (CTL) to specify the requirements, the way the modelling of some components is performed is similar to ours. Unfortunately many interesting problems (like direction of the flow of data) are not taken into account, and no complexity results (apart from experimental ones) are provided.

Our work can be placed somewhere in between this two. In particular our contributions are:

Formalisation : We provide a rigorous formalisation of the problem and analyse how to model different components in detail,

Reconfiguration as SAT : we show how the reconfiguration problem can be solved by means of incremental SAT solving,

NetSAT Prototype : we describe the important steps in the development of the NetSAT prototype (Section 6),

Complexity : finally, given the formalisation, we are able to provide some interesting complexity results (Section 7) that justify our implementation choices.

This work has to be considered as an introductory step to solving the problem: while developing the theory and the application, we faced many interesting question that we were unable to explore due to time constraint; we will try to present a quick overview of this ideas in the conclusive section.

2. General overview

This section is dedicated to give an overview of the domain of this work. We start by defining which components of the network we will take into account and we give a brief description of what their behaviour is.

In order to justify our choice, we will provide a short introduction on *what* kind of properties of the network we are interested in and we will introduce a few important modelling assumptions. We conclude the section by recalling some concepts from artificial intelligence that are important for the understanding of this work.

2.1. Components definition

A communication network is a network that allows the exchange of data between systems (that might be physically distant) without human intervention.

In the following we will use *network* to refer to computer networks but many ideas apply to other networks too (such as telephone networks).

A network is composed by active and passive components. As a rule of thumb passive components are those that don't need electricity to work, while active components do.

Even though passive components are vital for building a network, there's not much to say about them, since they don't require configuration, and their behaviour is usually not modifiable (even though they can be added or replaced). Passive components are wall plugs, cables, splitters, etc.

We will assume that the passive components are designed correctly (i.e. they are compatible with the network configuration) and we will focus mainly on active components.

Depending on the level at which we study the network, we can define different kinds of active components. We will take into account the following categories:

Host Is an active component that has at least a network interface and is reachable by an address,

Server Is a host providing some service,

Client Is a host wanting to access a service,

Router Is a host with more than one interface that receives packets from one interface and forwards them on another interface, depending on some rules that involve the destination of the packet,

Firewall Is a host that might drop packets passing through it. If it has the ability to keep track of active connections and make decisions based on the content of the connection we call it *Stateful Firewall*.

NAT / DNAT Is the technology used to modify the address/port of the destination/source of a packet. NAT is conceptually performed by routers.

We will work only with some information about the configuration of these components. In particular we are interested in:

- the address and default router of each host,
- the routing table of a router,
- the firewall rules of a firewall,
- the NAT table of a NAT,

2.2. Level of detail

There are many different levels of detail at which we could focus when talking about networking. We considered the ones presented in the TCP/IP ([18]) suite and decided to work at the Transport level. We will now shortly try to motivate this decision.

The TCP/IP suite is composed by the following layers:

- Application (HTTP, SMB, etc.)
- Transport (TCP, UDP, etc.)
- Internet (IPv4, IPv6, ICMP, etc.)
- Link (ARP, MAC, etc.)

The Link layer is the first to actually have addresses, making it a good candidate for our purposes. Unfortunately at this level we have different protocols depending on whether we are talking about wired or wireless clients; especially for wireless clients this means taking into account many possible standards.

In order to correctly represent the Application level it would be necessary to formalize at least one protocol. Assuming we consider one widespread protocol such as HTTP, we would still need to face compatibility dependency of the software installed both on the client and on the server. Moreover the final solution would be of little use if we considered only one protocol. Therefore we aimed at providing a solution that would be easily extensible to deal with this level, but not focused on it.

The Transport and Internet layers will be our focus. More precisely we will deal with IPv4 and TCP thus covering most of the LAN technologies in use. We will not use any specific feature of the TCP protocol, therefore everything expressed her can be used for UDP without modifications.

2.3. Policy overview

The main question we would like to answer is what configuration satisfies the policy. The policy is a collection of properties that must be independently satisfied by the network. A property states which clients can access a service (*reachability*) and which clients cannot access it (*unreachability*): *A client (or a group) should be able to reach a service if and only if it is defined in the policy.* Additionally we will provide a way to express some characteristic that we want the path taken by the packet to have.

The policy is satisfied if and only if all the properties are satisfied. If one property is not satisfied we want to know how to possibly modify (*reconfigure*) the network in order to solve the problem. Moreover, especially in case of unreachability properties, we would like to get a counterexample of the property, in order to get an idea of *why* the property is not satisfied.

2.3.1. Verification in a nutshell

For each property we need to look for the existence of a path that satisfies the following conditions:

- it starts from the client,
- it ends in the server,
- all firewall rules are satisfied,
- all routing rules are satisfied,
- all NAT rules are satisfied,
- it satisfies the additional conditions expressed by the property.

2.4. Modelling assumptions

Before presenting the formalization of this problem, we will introduce and justify some modelling assumptions that we made. In particular, we want to answer to the following question: Does time matters in our model?

We identified two possible solutions:

Timed: Time is an important component of this approach and events can influence each other. This allows us to consider multiple packets travelling in the network at the same time and how they influence the behaviour of the network.

Static: We only have one packet and this is the only thing we can modify.

Since TCP is session based, we might want to express sessions in our model. This would require a timed view in which the connection is established, used and finally closed. Using this kind of approach makes also sense in case we want to deal with stateful firewalls, or rules that deal with time (e.g. rules that limit the amount of packets per second).

Using a static view provides a great simplification of the model. We are not interested anymore to single packets but to data-flows that respect some "parameters". The downside is that we cannot model correctly stateful firewalls. We still can manage sessions by keeping track of the "state" of the connection in the packet but we have to face the problem of how to encode correctly a new "incoming" communication.

Lets introduce an example to clarify the problem:

A stateful firewall can look inside a session and act accordingly. FTP (File Transfer Protocol) works by exchanging commands through a particular TCP port (21), but when a client requires to transfer some data it opens an auxiliary port and tells the server to connect to it. The stateful firewall will recognise this command and allow the server to connect to the client on the specified port that is, otherwise, blocked.

This behaviour cannot be modelled by the static view.

The work presented here will be using a *static view* over a fixed configuration, taking into account a simplified TCP/IP packet. This means that we have two strong assumptions:

Determinism the same packet over the same component is always treated in the same way,

Independence packets do not interfere with each other.

These assumptions allow us to model the most relevant network components and, in general, we try to develop a framework that is generic enough to be applicable to other components as well (e.g. IPSec [14]).

2.5. Background knowledge

In the following work, we assume the reader to be familiar with a few areas of artificial intelligence such as propositional logic, SAT, quantified boolean formulas, planning and complexity analysis.

Propositional logic In this work we formalize most of the concepts as a propositional formula and then try to solve some problems related to this formula. Given a propositional formula, we say that it is *satisfiable* if there exists an assignment for its variables (a mapping of each variable to true or false) such that the formula evaluates to true. We call a formula *unsatisfiable* if such an assignment does not exists, or *valid/tautology* if it always evaluates to true. Lets call *literal* either a variable or its negation; a formula is in *CNF* if it is a conjunction of disjunction of literals. The problem of deciding whether a formula in CNF is satisfiable is called SAT; similarly, we call UNSAT and TAUT respectively, the problems of deciding whether a CNF formula is unsatisfiable or a tautology. A quantified boolean formula (QBF) is a propositional formula in which we quantify the variables with an existential ($\exists$) or universal ($\forall$) meaning. A QBF has the shape $Q_1x_1Q_2x_2...Q_nx_n.\phi$, where ϕ is a propositional formula, $Q_1,..,Q_n \in \{\exists, \forall\}$ and $x_1,..,x_n$ are the variables appearing in ϕ. A QBF is *true* if the satisfying assignments of ϕ satisfy the quantifications in the order: e.g. $\exists x_1 \forall x_2.\phi$ is satisfied if and only if *there exists* a value for x_1 such that *for all* values of x_2 ϕ evaluates to true.

SAT and QBF are covered in any introductory book in complexity, throughout this work we use [21] as reference.

Planning Planning provides the tools for modelling problems in which we are interested in a series of "actions" needed to reach a "goal" from an "initial" situation. To do this, we define *states* and *operators* between states. We are interested in propositional planning, therefore each state is associated with the same set of propositional variables but with a different assignment for these variables. We call the initial situation *initial state* and, since we might have more than one goal state, we represent the goal as a propositional formula expressing some property of the set of goal states. Operators allow us to go from one state to the other and will be defined more in detail in section 3.4. Each operator defines an action, and we call the set of available operators *domain*, and a sequence of (possibly repeating) operators a *plan*; we are interested in *sequential* planning, that is at each time we can apply only one operator (as opposed to parallel planning). A good introduction to planning can be found in [23].

Complexity When defining a problem, we are interested in understanding how "fast" we can solve instances of growing size. We will apply in section 7 the classical framework of worst-case complexity analysis to our problem; therefore, we assume familiarity with concepts such as reductions, hardness, completeness, and the classes NP and coNP. We will also need the second level of the polynomial hierarchy Σ_p^2. This class is sometimes denoted with the oracle notation NP^{NP} or with the related QBF problem that is complete for the class: $\exists\forall$ [21]. Finally, we will also provide some results from the framework of parameterized complexity. We provide a short introduction to the relevant definitions in section 7; for a brief and update introduction to the area we suggest [15].

3. Translation

In this section we present the details to represent the network components in a propositional way. We will first introduce the key idea of *Packet* that is, in practice, the alphabet on which we express our formulae. Afterwards, we will present an example network that will help us through the explanation of various modelling problems. This should also help the reader get a better idea of the problem's domain before introducing the translation of the various components as propositional formulae. Finally, we conclude the section by presenting the encodings from a planning perspective.

3.1. The Packet

The packet contains all the information on which we can work; in particular the packet encodes the following informations:

- Source IPAddress (32bit)
- Destination IPAddress (32bit)
- Source Port (16bit)
- Destination Port (16bit)

The encoding of the packet will thus consist of the 4-tuple PKT=(SrcIP,DstIP,SrcPort,DstPort) (96 bits) and a set of additional variables indicating the position of the packet in the network Pos.[1].

3.1.1. Encoding numbers

Before entering into the details of the translation, we should discuss the different ways in which we can encode numeric values representing addresses, ports and positions:

Unary: We introduce one boolean variable for every possible value (N). This solution makes the actions simple to write since we need to check and change only two variables. The downside is that it is not applicable to big values, and that it allows to encode more than one value simultaneously: we need to add $\mathcal{O}(N^2)$ constraints to the problem to avoid this behaviour.

Binary: Encoding the value in binary works well also with big numbers ($log_2 N$) and prevents multiple values. Unfortunately more encoding is required when changing the values. Fortunately we don't deal with complex operations, but only with assignments (and equalities). This means that in each operation we need to express at most $log_2 N$ atomic values for each change.

[1] A rough estimation of the bits needed can be made by considering that we will use 1 additional bit per network component and up-to 6 network components per address of each host.

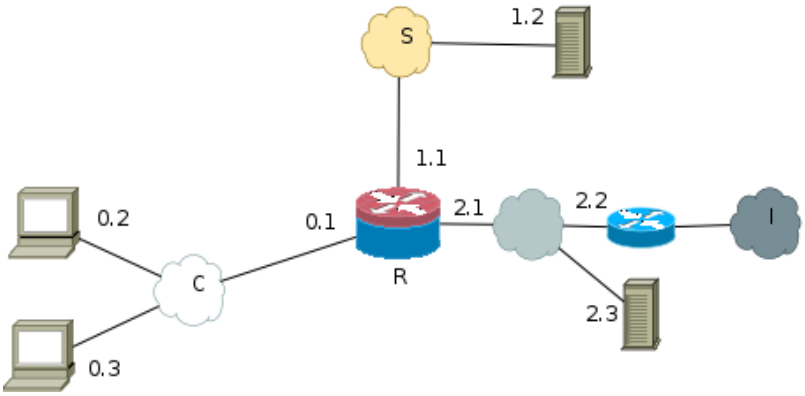

Figure 3.1.: Example Network

Mapping: If we are interested only in few values over a big range, we can "map" these values with values in a smaller range. For example if we know that the most common TCP ports are 22,25,80,110, then we can encode them as 0,1,2,3. In general we can take our configuration and map all used port numbers to smaller numbers. This method requires some pre-processing in order to guarantee that all occurrences of a number are mapped correctly, but can significantly reduce the number of variables in use.

In this translation we are using unary encoding for the Positions, since they are supposed to be a small number, and binary encoding for Addresses and Ports. In the following, to keep the notation simple, we will write $\leftrightarrow$ when testing equality both for unary and binary values.

3.2. Translating real world components

We now introduce our example network (Fig. 3.1) that will be useful also later one, when talking about policies and reconfiguration. This network contains some hosts, one router (with Firewall and NAT rules) and an Internet access[2].

Lets now shortly introduce each relevant component of our network:

3.2.1. The router "R"

This router allows the clients of the subnet C to access the servers of the subnet S and the internet (represented by the subnet I).

It has the following rules:

Routing tables: As you can see, we do not specify a gateway for the subnets to which the router is directly connected (0.0/8, 1.0/8 and 2.0/8), but we only define the default gateway as 2.2: this will allow R (and the clients) to reach the internet. [3]

[2]For brevity I will use 16bit addresses. Thus, the netmasks are from 0 to 16, with /16 indicating a specific host.
[3][18] is our main reference for the networking technical details.

Dest	NetMask	Gw
0.0	/8	*
1.0	/8	*
2.0	/8	*
*	*	2.2

Firewall rules: The last rule (denoted by $*$) is the default rule. Firewall rules are evaluated on a first-match base therefore, if none of the other rules matches the current packet, the last rule is used. The rules in red represent a mistake in the configuration since their order has been inverted. This will cause the server 1.2 to be unreachable from the client network.

Condition	Action
SrcIP = 0.0/8 and DstIP = 0.1/16 and DstPort = 22	Accept
SrcIP = 0.0/8 and DstIP = {0.1/16,1.1/16,2.1/16}	Reject
SrcIP = 0.0/8 and DstIP = 1.0/8	Reject
SrcIP = 0.0/8 and DstIP = 2.0/8	Reject
SrcIP = 0.0/8	Accept
DstIP = 1.2/16 and DstPort = 80	Accept
*	Reject

NAT Rules: The meaning of these rules is that (e.g.) all the packets meeting the conditions of the first rule, will have the destination IP address changed into 1.2. The purpose of these rules is to allow the web server 2.1 to be accessible from the internet and the clients of the subnet C to connect to the internet.

Type	Condition	Action
PreRouting	DstIP = 2.1/16 and DstPort = 80	DstIP = 1.2
PostRouting	SrcIP = 0.0/8 and DstIP != 1.0/8	SrcIP = 2.1

3.2.2. Clients and Server

Routers are usually the more complex components in a computer network; for this reason, and to try to keep our example simple, we will assume that the clients have no firewall or NAT rules and that their default router is R. The server 1.2 will have a firewall policy that allows access only to port 80, and it will use R as default gateway. Finally, no configuration is specified for the server 2.2, whose default router will be 2.3.

Host	Default gateway	Note
0.2, 0.3	0.1	
1.2	1.1	Allow only access to port 80
2.2	2.3	

3.2.3. Expanded network

Previously (in section 2) we considered components that were disjoint one from the other. As the above example shows, this is not usually the case when we look into a network at the IP level: everything has an Address, therefore everything is a Host.

We call *Expanded* network the model of a network in which each host has been expanded into its network components.

After expanding the NAT and Firewalls of the example network, and building links to express the default gateways, we obtain the expanded network of figure 3.2. Blue arrows indicate a routing rule in which the next router is specified (also known as *next-hop*); in our example the only rules of this kind are default gateway rules.

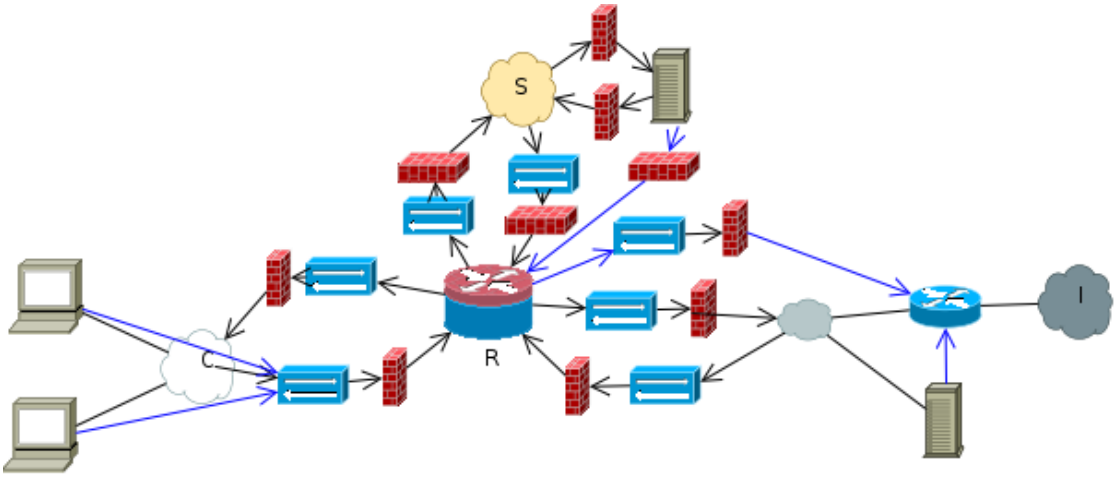

Figure 3.2.: Expanded Network

A host that has Routing-, FW- and NAT-Rules will be expanded into 5 components per each address. In particular for each Address the firewall and NAT will be duplicated, in order to have an outgoing and an incoming path. Moreover a direct path to the next-hops will be added. Given an Address belonging to a subnet, the host looks like:

$$Subnet \rightarrow NAT \rightarrow FW \rightarrow Router \rightarrow NAT \rightarrow FW \rightarrow Subnet$$

Motivation The first motivation of this expansion is to be able to write simple rules that model the behaviour of each network component independently. When talking about the Packet, we describe the addition of a variable *Pos*, indicating the position of the packet in the network. Since this variable refers to one of this components it is easier to evaluate the specific rules of the component on a case by case base. On a side note, we expect it to be easier to expand this framework to higher and lower levels, by simply adding more components to the network and connecting them accordingly. The second motivation of this expansion, is to be able to distinguish between packets arriving to and leaving a particular host. In real networks this information is encoded at a lower level[4], but we lose this information when taking into account only the IP level. This expansion does not allow us to take into account full directionality, but we think that it is a good compromise between complexity and expressivity.[5]. On a side note, this is one of the modelling problems that is not handled by configChecker ([5]).

3.3. Network Components

We now formalise the behaviour of the network components. Before doing so, we recall that a host is any network device that has at least an (IP)Address; we then build the expanded network by splitting each host into its network components:

[4]By the MAC address of the packet

[5]By full directionality we mean the ability of acting on the packets depending on which are the input and output networks. For a more detail explanation of the problem, refer to appendix A.1

- 1 Router
- at most 2 NAT (per address)
- at most 2 Firewall (per address)

As justified in the previous section, we can now describe each component behaviour independently. We do so by defining a characteristic formula: a propositional formula that evaluates to true when the component behaves correctly.

Given a generic network component C, it will process packets whose Position matches its own:

$$\delta_{C_*} \doteq (Pos \leftrightarrow C) \rightarrow \delta_*$$

where δ_* encodes the characteristic formula of the component.

3.3.1. Router

We define two kinds of routers: forwarding and non-forwarding. We represent any host as a non-forwarding router in order to have a simpler encoding of the hosts. In general when we talk about Routers, we mean forwarding routers.

Lets take a generic Router R such that:

- It has N IPAddresses $(IP_1, .., IP_n)$.
- In the routing table it has k ordered routing rules
- Its routing rules are in the form: (Destination, Netmask, GatewayIP)
- It can be enabled to forward packet

Thus, its behaviour would be to

- Accept packets that are meant for it
- Send/Forward packets

We can encode it as follows:

$$\delta_H = \mu \leftrightarrow \neg\nu$$

either the packet is for it (μ) or it is sending the packet (ν):

$$\mu = \bigvee_{i=1}^{N} (DstIP \leftrightarrow IPAddress_i)$$

with N number of IPAddresses related to the Router; and

$$\nu = (\bigvee_{i=1}^{N} (SrcIP \leftrightarrow IPAddress_i) \vee forwading_R) \rightarrow \delta_R$$

meaning that the packet goes through the routing tables if either it is the router that is sending it or it is enabled for forwarding.

We encode the routing table by encoding the routing rules as *condition* $\rightarrow$ *action*:

$$\delta_R = \bigwedge_{i=1}^{k} \phi_i$$

$$\begin{aligned}
&\phi_1 = \psi_1 \rightarrow \sigma_1 \\
&\phi_2 = \neg\psi_1 \wedge \psi_2 \rightarrow \sigma_2 \\
&... \\
&\phi_k = \textstyle\bigwedge_{i=1}^{k-1} \neg\psi_i \wedge \psi_k \rightarrow \sigma_k
\end{aligned}$$

where ψ encodes the condition that the packet matches the rule, namely that after applying the Netmask to the IPAddress, the result equals the Destination:

$$\psi_i \approx (DstIP \wedge R_i Netmask) \leftrightarrow R_i Destination$$

more precisely

$$\psi_i = \bigwedge_{j=1}^{32} (DstIP_j \wedge R_i Netmask_j) \leftrightarrow R_i Destination_j$$

σ is the action that "moves" the packet from the router to the next node. Note that how we decide which is the next node is greatly influence by how we model the network. For now lets assume that n^* is the "correct" next node.

$$\sigma_i \approx (Pos' \leftrightarrow n_i^*) \wedge (PKT' \leftrightarrow PKT)$$

where the primed version of the variables means (from now on) the value of the variable at the next time step. A particular case of these rules is the default router, that has $\psi_k = \top$ since it matches all the packets.

Choosing n^* If the matched routing rule doesn't specify a gateway, this means that we are going to send the packet to the Subnet (see section 3.3.4), therefore n^* is the next component (in the expanded network) of the router leading to the subnet. Otherwise, if the gateway is specified, n^* corresponds to a network component of the expanded model of the gateway. In general n^* is the next component to which the arrow in the expanded network is pointing to (ref. fig. 3.2).

3.3.2. Firewall

In the expanded network we duplicate each firewall creating two new components (per address): F_{IN} and F_{OUT}. This division will allow us to know what component is n^*. Note that this division does not apply to the matching of the rules, since we do not want to take into account directionality, all rules are checked on both sub-firewalls. It is important to stress that $F_{IN} \approx F_{OUT}$ with the only difference of n^*.

Recall, from the example, that the rules are in the form (Condition, Action), thus the translation of firewall F with k ordered FirewallRules is:

$$\delta_F = \bigwedge_{i=1}^{k+1} \phi_i$$

where:

$$\phi_1 = \psi_1 \rightarrow \sigma_1$$
$$\phi_2 = \neg\psi_1 \wedge \psi_2 \rightarrow \sigma_2$$
$$...$$
$$\phi_k = \bigwedge_{i=1}^{k-1} \neg\psi_i \wedge \psi_k \rightarrow \sigma_k$$

where ψ_i encodes the condition, that might take into account equality of any field of the packet with single or multiple values.

If the i-th rule has Action=Accept then:

$$\sigma_i = (Pos' \leftrightarrow n^*) \wedge (PKT' \leftrightarrow PKT)$$

but if Action=Deny then:

$$\sigma_i = (Pos' \leftrightarrow 0)$$

this rule will be discussed in more detail later on, but the idea is to move the dropped packets to a particular position (empty position), in order to be able to verify this behaviour in the policy.

The default rule is included by:

$$\phi_{k+1} = \bigwedge_{i=1}^{k} \neg\psi_i \rightarrow \sigma_{k+1}$$

3.3.3. NAT

As for the Firewall, also for the NAT we will differentiate the direction of the packets. Nevertheless, here the difference is remarkable since NAT rules are partitioned into pre-routing and post-routing and they make sense only when evaluated before (or after) applying the routing rules.

Given a NAT N we define the two new NAT, N_{IN} and N_{OUT}, such that the NAT Rules of N are totally partitioned in those of N_{IN} and N_{OUT} depending on whether they are pre-routing or post-routing rules respectively. NAT rules are also expressed as (Condition,Effect), thus the translation of a NAT $N_{IN/OUT}$ with k ordered NATRules is:

$$\delta_{N_x} = \phi_k \wedge (Pos' \leftrightarrow n^*)$$

with

$$\phi_1 = \psi_1 \rightarrow \sigma_1$$
$$\phi_2 = \neg\psi_1 \wedge \psi_2 \rightarrow \sigma_2$$
$$...$$
$$\phi_k = \bigwedge_{i=1}^{k-1} \neg\psi_i \wedge \psi_k \rightarrow \sigma_k$$

where ψ_i encodes the condition that might take into account equality of any field of the packet with single or multiple values.

σ is different from the previous cases:

$$\sigma_i = PKT' \leftrightarrow PKT_{nat}$$

with PKT_{nat} equal to PKT except for the fields that are modified by the action.

It is quite clear that Router, Firewall and NAT encode rules that are really similar to each other. This will become even more evident when presenting the corresponding planning problem.

3.3.4. Subnet

The Subnet component was introduced to make the computation of n^* easier. When a router wants to forward a packet to a subnet, the packet should be sent to *all* the components of the subnet at the same time and only the right recipient should process it. By introducing the Subnet we avoid having non-deterministic actions and every packet is forwarded to at most one other node. This simplifies the model since each component can be described more independently of the others and, moreover, it prevents the routers connected to a subnet from forwarding packets that are meant to one of the other devices in the subnet.[6]

We characterise a subnet S by a range of IP addresses: all the hosts with an IPAddress in the range are then connected to the subnet.[7] A subnet can then be translated as:

$$\delta_S = \bigwedge_{i=1}^{N}((H_i IPAddress \leftrightarrow DstIP) \rightarrow n_i^*)$$

where N is the number of hosts connected to the Subnet, n_i^* is the i-th host and $H_i IPAddress$ its address.[8]

3.4. Viewing the components in a Planning perspective

By looking at the δ_* formulae of router, firewall and NAT we can notice that all of them are in the form $Condition \rightarrow Effect$. This fits well with the definition of *Operator*s of [23] :

Definition 2.9 Let A be a set of state variables. An operator is a pair $\langle c, e\rangle$ where c is a propositional formula over A (the precondition), and e is an effect over A. Effects over A are recursively defined as follows:

1. a and $\neg a$ for state variables $a \in A$ are effects over A.
2. $e_1 \wedge ... \wedge e_n$ is an effect over A if $e_1, ..., e_n$ are effects over A (the special case with $n = 0$ is the empty effect).
3. $c \triangleright e$ is an effect over A if c is a formula over A and e is an effect over A.
4. $e_1|...|e_n$ is an effect over A if $e_1, ..., e_n$ for $n \geq 2$ are effects over A.

$e_1|...|e_n$ expresses a non-deterministic effect, since we don't have non-determinism we don't need to take this into account. Conditional-effect $(c \triangleright e)$ are useful to express more compactly

[6] As a more technical side note, you can think of the Subnet component as a Switch, as opposed to the previous (non-deterministic) behaviour that modelled a Hub. We don't need such artefact if we consider the ARP level in our model.

[7] This is the simplest approach that completely abstracts from the lower layers that define the topology of the network.

[8] A host might be connected with more than one address to the same subnet. In that case we will have one rule per address.

(or intuitively) the behaviour of an operator, but they are not necessary, since we can always write an equivalent normal form in which we only have atomic conditional effects, meaning that conditional effects are not nested and the effect is either $\top$ or an atomic effect (section 2.3.3 [23]). This in turn translates into the possibility of having more complex conditions in the operator and not having conditional effects at all[9]. In the following we will make use of conditional effects to express more compactly the operators of every component:

Router:

$$\langle Pos_H \wedge \neg\mu \wedge \nu, \bigwedge_{j=1}^{k}((\bigwedge_{i=1}^{j-1} \neg\psi_i \wedge \psi_j) \triangleright (\neg Pos_h \wedge Pos_{n,j^*})\rangle$$

where Pos_H is the component position (encoded in unary). ψ is conceptually the same as defined earlier but since we encode the IP Address in binary, applying the netmask simply means comparing the first N bits, where N is the Netmask value.

$$\psi_i = \bigwedge_{j=0}^{R_i Netmask_j} DstIP_j \leftrightarrow R_i Destination_j$$

μ and ν encode the information of whether the packet is for us (μ) or we are sending/forwarding it (as defined earlier).

$$\mu = \bigvee_{i=1}^{N}(DstIP \leftrightarrow IPAddress_i)$$

with N number of IPAddresses related to the Router. We will consider the ability to forward a packet to be a structural property of the router, i.e. for forwarding routers $\nu = \top$ otherwise:

$$\nu = \bigvee_{i=1}^{N}(SrcIP \leftrightarrow IPAddress_i)$$

Firewall:

$$\langle Pos_H, ((\bigvee_{j=1}^{k}(\bigwedge_{i=1}^{j-1} \neg\psi_i \wedge \psi_j)) \triangleright Pos_{n_j^*}) \wedge \neg Pos_H\rangle$$

The conditional effect is defined only if the j-th rule is an Accept rule. This way if the packet matches a Deny rule, it will go to an "empty" position (all Pos_x are 0). This will help us to identify the paths that do not reach their destination.

Note that the same operator can be expressed by using a conjunction of conditional effects. In that case instead of having only one conditional-effect we would have k.

In general ψ can be any arbitrary propositional formula, but we will start considering only equalities between a field in the packet and a value. This approach makes range comparisons very verbose but is expressive enough to be a good starting point.

[9] This can lead to an exponential blow-up of the number of the operators

NAT:

$$\langle Pos_H, \bigwedge_{j=1}^{k}((\bigwedge_{i=1}^{j-1}\neg\psi_i \wedge \psi_j) \triangleright e_{j,PKT}) \wedge Pos_{n^*} \wedge \neg Pos_H\rangle$$

where $e_{j,PKT}$ modifies the PKT according to the j-th rule. Remember we divided the rules between PRE- and POSTROUTING rules, so we don't have to encode this condition. Similarly to the firewall, we might want to encode as action an arbitrary propositional formula. For now we will consider only equality constraints that are, nevertheless, the most common actions in a NAT.

Subnet:

$$\langle Pos_H, \bigwedge_{j=1}^{k}((\bigvee_{i=0}^{n_j}(DstIP = IPAddress_{j,i})) \triangleright (\neg Pos_H \wedge Pos_{n_j^*}))\rangle$$

with k hosts connected to the subnet, each of them with n_j IPAddresses.

3.4.1. Limitation of this approach

We now want to present some limitations of the planning solution that has been shown so far; to do so, we briefly introduce the simplest property that we want to test: simple reachability. The rest of the properties will be properly introduced in section 4.

What we described so far is the *domain* of the problem. The domain can be automatically generated by specifying the model of the network. We are going to define now the initial state and the goal. These informations are part of the property that we want to test:

Initial state: Defines the packet and the starting position (Client)

Goal state: Defines the goal position (Server)

Simple reachability simply means that we want to test whether the given client can reach the server with a particular packet.

This property can be tested with the framework described so far, but the counter part of this property (unreachability) only works in theory. In fact to test unreachability we need to show that there is *no* plan from the client to the server, and to do so, we need a complete algorithm. Algorithms for planning are usually complete under the assumption of termination: if the algorithm terminates, we are provided with all available solutions (if any). Unfortunately, performing the termination test is usually impractical and, therefore, we are usually unable to show the non-existence of a plan.

3.4.2. Improving the PDDL translation

We encoded our planning problem in PDDL and tried to solve the simple reachability property with different planners ([22],[29]). Unfortunately not all planners support conditional-requirements and disjunctive preconditions; moreover, the approach described so far, doesn't give much information when a packet cannot reach the destination. I will thus briefly show how we tried to solve these two problems.

Empty State As discussed when talking about the firewall rules (3.3.2), the best way to know why a packet hasn't been delivered is to add some sort of empty state, so we can test whether the packet reached this state and know which was the last node to process it.

Implementing this means that the effect of being in any component is always (at least) leaving the component (i.e., $\neg Pos_H$). For example the router becomes:

$$\langle Pos_H \wedge \neg\mu \wedge \nu, \bigwedge_{j=1}^{k}((\bigwedge_{i=1}^{j-1} \neg\psi_i \wedge \psi_j) \triangleright Pos_{n_j^*}) \wedge \neg Pos_H \rangle$$

The goal to see what component "discarded" the packet would be:

$$\bigwedge_{i=0}^{N} \neg Pos_i$$

This rule allows us to detect firewall rules dropping the packet, or routers for which no rule was defined for the particular packet. Note that this doesn't allow us to detect loops in the configuration. Moreover, this approach assumes that only one action is performed by each component; while this is currently true, we must take this into account before extending the framework.

3.4.3. Avoiding advanced requirements

Support for conditional-effects is not widespread among solvers. Therefore, even though they allow us to express the actions more compactly, we will redefine the components without using them. The standard way of removing conditional effects might bring an exponential blow-up in the number of operators. We can show that in our case most of the operators resulting from the standard translation would have unsatisfiable preconditions, [10] therefore we are able to avoid using conditional effects by writing a linear number of operators: one for each different rule.

Disjunctive-precondition are used to match ordered rules. Unless the rules are disjointed[11], we cannot remove them. Nevertheless, we try to limit their usage to only this particular case.

In the following we show the *improved* translation of the network components:

Router

$$\begin{aligned}
&\langle Pos_H \wedge \neg\mu \wedge \neg\nu, \neg Pos_H \rangle \\
&\langle Pos_H \wedge \neg\mu \wedge \nu \wedge \psi_1, Pos_{n_1^*} \wedge \neg Pos_H \rangle \\
&\langle Pos_H \wedge \neg\mu \wedge \nu \wedge \neg\psi_1 \wedge \psi_2, Pos_{n_2^*} \wedge \neg Pos_H \rangle \\
&... \\
&\langle Pos_H \wedge \neg\mu \wedge \nu \bigwedge_{i=1}^{k-1} \neg\psi_i \wedge \psi_k, Pos_{n_k^*} \wedge \neg Pos_H \rangle
\end{aligned}$$

Firewall: In the previous translation we took into account only *Accept* rules. Here we write an action also for Reject.

$$\begin{aligned}
&\langle Pos_H \wedge \psi_1, e_* \rangle \\
&\langle Pos_H \wedge \neg\psi_1 \wedge \psi_2, e_* \rangle \\
&... \\
&\langle Pos_H \bigwedge_{i=1}^{k-1} \neg\psi_i \wedge \psi_k, e_* \rangle \\
&\langle Pos_H \bigwedge_{i=1}^{k} \neg\psi_i, e_{default} \rangle
\end{aligned}$$

[10] Consider the NAT: the rules are of the type *if* you did not match the previous rule and match this one *then* apply this one. This means we apply only one rule. When we translate the operator we don't stop the first time a condition is matched, but we keep expanding the operator for any possible combination of cases (2^N). Thus we will end up with an operator that has as precondition : $\psi_N \wedge (\psi_{N+1} \wedge (\neg\psi_{N_0} \wedge .. \wedge \neg\psi_N))$ that is unsatisfiable.

[11] See appendix A.2

with $e_* = \begin{cases} \neg Pos_H \wedge Pos_{n^*} & \text{if it is an Accept rule} \\ \neg Pos_H & \text{otherwise} \end{cases}$

and $e_{default}$ encodes the default behaviour of the firewall.

NAT

$$\langle Pos_H \wedge \psi_1, e_{1,PKT} \wedge Pos_{n^*} \wedge \neg Pos_H \rangle$$
$$\langle Pos_H \wedge \neg\psi_1 \wedge \psi_2, e_{2,PKT} \wedge Pos_{n^*} \wedge \neg Pos_H \rangle$$
$$...$$
$$\langle Pos_H \textstyle\bigwedge_{i=1}^{k-1} \neg\psi_i \wedge \psi_k, e_{k,PKT} \wedge Pos_{n^*} \wedge \neg Pos_H \rangle$$
$$\langle Pos_H \textstyle\bigwedge_{i=1}^{k} \neg\psi_i, \wedge Pos_{n^*} \wedge \neg Pos_H \rangle$$

where $e_{j,PKT}$ modifies the PKT according to the j-th rule. Note that the last rule states that NAT are never dead ends, i.e. the packet always moves to the next position.

Subnet

$$\langle Pos_H \wedge DstIP = IPAddress_{0,0}, (\neg Pos_H \wedge Pos_{n_0^*})) \rangle$$
$$\langle Pos_H \wedge DstIP = IPAddress_{0,1}, (\neg Pos_H \wedge Pos_{n_0^*})) \rangle$$
$$..$$
$$\langle Pos_H \wedge DstIP = IPAddress_{0,m_0}, (\neg Pos_H \wedge Pos_{n_0^*})) \rangle$$
$$...$$
$$\langle Pos_H \wedge DstIP = IPAddress_{k,m_k}, (\neg Pos_H \wedge Pos_{n_k^*})) \rangle$$
$$\langle Pos_H \wedge \textstyle\bigwedge_{j=0}^{k} \neg(\bigwedge_{i=0}^{m_j} DstIP = IPAddress_{j,i}), \neg Pos_H \rangle$$

with k hosts connected to the subnet, each of them with m_j IP addresses. This encoding assumes that IP Addresses are unique in the subnet. Note the last rule that sends the packet to the empty state if no such address exists in the subnet.

3.5. Section summary

In this section we introduced the key concepts of this work. We started by talking about our alphabet (the Packet) and then provided an example network to clarify the domain at hand; after introducing the expanded network, we presented two approaches for translating the network components: propositional and planning.

Even if, in our experiments, the planning approach turned out to be inadequate for more complex properties, we think it is really interesting to see how simple and elegant the translation can be for simple reachability problems. This simplicity motivated us to base our final solution on planning as SAT: working on the SAT encoding of the planning problem.

4. Policy specification

The verification problem requires two components: the network and the policy. In this section we will focus on the policy.

We define a *Policy* as a set of *network properties* that must all be satisfied by the network configuration. In this section we will explore what kind of properties we are interested in, i.e. the expressivity of the network properties.

To give a rough idea of what are we talking about, lets take our example from section 3.2. A possible policy for this network could be:

1. Nobody (except (Subnet C on port 22) and the router itself) can reach the router,
2. Everybody can reach port 80 on S
3. Connections to the router on port 80 should be forwarded to 1.2
4. The external router should not be able to reach the clients (Subnet C)
5. Nobody should be able to reach S (except that on port 80)

It should be clear how the concept of "reachability" plays a key role in the policy. The only rule that is not concerned with reachability is 3., that expresses some properties of the path taken by some packets.

We will start by defining the *basic reachability problem* as the most expressive property that we are interested in and from there we will work out some simpler sub-properties. Afterwards, we will show the encoding of these properties into classical logic problems; finally, we conclude with the encoding of the properties in the planning perspective.

4.1. Network properties

A Network property is a "formula" that allows us to express the basic reachability problem.

Given:

- a network configuration C [1]
- an initial position Pos_0,
- a formula characterising a non-empty set of initial packets τ,
- a formula characterising the path $VALID$
- a final (goal) position Pos_G,
- and an integer n

the basic reachability problem (BRP) corresponds to the following decision problem:

[1] see section 5 for the formal definition of C.

Definition BRP Is it possible in the network C for all the packets p (s.t. $p \models \tau$) starting from Pos_0 to reach Pos_G in n steps (or less) and satisfy the condition $VALID$?

From the BRP we define simpler problems both in term of complexity and expressivity:

P1 Simple (Un)Reachability

P2 Element traversal

P3 Full (Un)Reachability

P4 Quantified (Un)Reachability

P5 Quantified element traversal (= BRP)

The definition of these problems is the same, but we limit the possible values for τ and $VALID$ as shown in the following table:

	τ	$VALID$
P1	1 Specific value for each field	$\top$
P2	1 Specific value for each field	Arbitrary over $Visited_i$ and PKT_n
P3	$\top$	$\top$
P4	Arbitrary over PKT_0	$\top$
P5	Arbitrary over PKT_0	Arbitrary over $Visited_i$ and PKT_n

where PKT_n represents the final configuration of the packet when it reaches the destination and $Visited_i$ represents whether the i-th component was part of the path or not.

It should be clear that all these problems are particular cases of P5 that is exactly the BRP.

We define the *unreachability* problem by simply changing the quantification of the BRP: "Is there a packet (s.t. $p \models \tau$) in the network C starting from Pos_0, reaching Pos_G in n steps (or less) and satisfying the condition $VALID$? "

We define the sub-problems for unreachability by applying the same restrictions as above. We will not distinguish between reachability and unreachability problems, unless it is necessary. In general, when we talk about a problem, we refer to the reachability version.

Lets briefly summarise the properties defined by each of this problems:

P1 The first problem is the one we already introduced in section 3.4.1: given a single packet and a starting position, we want to know if it can (or cannot) reach a given goal position.

P2 The second problem extends P1 by allowing us to analyse some property of the path taken by the packet: "Is the path between A and B going through the component C ?" or "Is the packet at the destination the same as at the source?". This allows us to know which elements are involved in the communication: maybe we want to remove them, or we know that there is some problem with connections passing through C or through components of a particular type[2]. The ability to express $VALID$ over PKT_n allows us to check the final value of the packet and verify, for example, that the NAT is behaving as expected.

P3 The third problem is useful when dealing with unreachability: we want to be sure that a given host cannot reach (with any packet) another host.

P4,P5 The fourth and fifth problems are simple extensions of P1 and P2 in which we consider an arbitrary **set** of packets having some property. This allows us to consider ranges of addresses/ports and exceptions (e.g., none but port 80).

[2]E.g. some VoIP systems don't work behind a NAT

What we are not interested to express with the BRP is information concerning:

- packet alteration along the path: e.g. "what changes are performed by each single network component?",
- IP level design properties: loops, duplicated addresses, reserved address etc.

4.2. Encoding the properties

Now that we know what properties we want to test, we can look for suitable encodings. In particular the encodings will also give us an idea of the complexity of the problems.

4.2.1. Properties definable via truth evaluation

We can easily solve problems **P1** and **P2** by reducing them to truth evaluation of a propositional formula.

For an n *big enough*, we define:

- I_0 to contain the initial state: $Pos_0 \wedge PKT_0$, with PKT_0 containing the information of the initial packet.
- G_n to represent the goal of reaching Pos_G in n steps.
- $R(0, n)$ to encode the transition relations from time 0 to time n

If $\phi_{P1} = I_0 \wedge R(0, n) \rightarrow G_n$ evaluates to true, it means that there's a path from Pos_0 to Pos_G of length $\leq n$ for the packet.

Since the network is deterministic, its behaviour is completely defined by $I_0 \wedge R(0, n)$: we define all variables at time $n + 1$ by unit propagation from time n.

If ϕ_{P1} is false then there are three possibilities:

- if the packet is not moving any more but it didn't reach the goal position, it could either be in the empty position or on another component;
- if the packet is still moving we might be using a low value of n or
- there might be some other problem (like the packet being in a loop).

The last two problems are strictly related and are solved by defining an upper bound for n that in the worst case will be fixed to a constant $c = 1300$[3] that is related to a characteristic of the IP protocol.

For P2 we need to add the $VALID$ constraint:

$$\phi_{P2} = I_0 \wedge R(0, n) \rightarrow (G_n \wedge VALID)$$

Where $VALID$ could, in principle, be an arbitrary formula, but (as defined before) we will limit it to formulas over PKT_n and $Visited_i$[4].

Similarly as for P1, the value of G_n and $VALID$ are completely defined by $I_0 \wedge R(0, n)$, leading to a simple truth evaluation.

For the particular case of P1 (and P2), the unreachability is defined as the negation of the reachability: i.e., we want $\neg\phi_{P1}$ (resp. $\neg\phi_{P2}$) to be true.

[3] see section 7.3 for the justification of this value and the definition of a possible lower value.

[4] Among the other things, we take care of setting $Visited_i$ in the relation $R(0, n)$.

4.2.2. Properties definable as SAT

P1 and P2 deal with checking properties over a single well-defined packet but all the other properties are interested in a "set" of packets that share a particular characteristic.

We will present immediately how to encode the BRP (P5) and then show how to express P4 and P3

To solve the basic reachability problem we verify that the following formula is a *tautology*:

$$\phi_{BRP} = \forall PKT_0.(\tau(PKT_0) \wedge Pos_0 \wedge R(0,n)) \rightarrow (Pos_G \wedge VALID)$$

that is equivalent to check whether $\neg\phi_{BRP}$ is UNSAT.

A simple way of "reading" this formula is: "If p belongs to the set of packets that we are interested in ($p \models \tau$), it starts from Pos_0 and all the network components behave correctly ($R(0,n)$) then it must reach Pos_G and the path must be valid."

The usual restrictions on $VALID$ and τ apply.

Verifying unreachability is, contrary to P1 and P2, a different problem from reachability. In fact we need to show that the following is a tautology:

$$\phi^{-}_{BRP} = \forall PKT_0.(\tau(PKT_0) \wedge Pos_0 \wedge R(0,n)) \rightarrow \neg(Pos_G \wedge VALID)$$

i.e., the following is UNSAT:

$$\neg\phi^{-}_{BRP} = \exists PKT_0.(\tau(PKT_0) \wedge Pos_0 \wedge R(0,n) \wedge Pos_G \wedge VALID)$$

Recalling the definition of P3 and P4, we see that we can obtain P3 by simply defining $\tau = \top$ and $VALID = \top$ and P4 by assigning $VALID = \top$ and τ as a formula expressing the set of packets that we are interested in.

An interesting feature of this encoding is that, if $\neg\phi_{BRP}$ is satisfiable, then we can read a counter example by interpreting its model and obtain useful information on where the problem might be (of course, the same applies for $\neg\phi^{-}_{BRP}$).

4.2.3. Extending the quantification

The definition of the BRP requires a single initial position Pos_0. Suppose we want to verify that all the subnet 192.168.0.0/16 can access 10 particular ports on a given server. Encoding this by means of the BRP would require expressing (in the worst case) one rule for every of the 65000 initial position. To avoid this, we will extend the quantification over the initial position Pos_0. To correctly characterise this problem we need to introduce an auxiliary formula σ relating each possible initial position with the IP addresses belonging to that host:

$$\sigma = \bigwedge_{i=0}^{N}((Pos_0 \leftrightarrow Pos_i) \leftrightarrow ((SrcIP \leftrightarrow IP_{i,0}) \vee ... \vee (SrcIP \leftrightarrow IP_{i,k}))) \wedge \bigvee_{i=0}^{N} Pos_i$$

with $IP_{i,j}$ being the j-th IP-Address of the network component Pos_i. The final disjunction expresses the need for one of the starting positions to be true. [5]. Note that in the extended network we have many components for each host, nevertheless only the router associated with each host should be considered as valid starting position.

We call this problem extended reachability problem (BRP^*), and we encode it as :

[5]Since we encode Pos in unary, we need to add an axiom that says that exactly 1 Pos can be true at time 0. $R(0,n)$ is defined in such a way that it is enough to enforce this condition only at time 0.

$$\phi_{BRP^*} = \forall PKT_0 \forall Pos_0.(\tau'(PKT_0, Pos_0) \wedge Pos_0 \wedge R(0,n)) \rightarrow (Pos_G \wedge VALID)$$

where $\tau' = \sigma(PKT_0, Pos_0) \wedge \tau(PKT_0)$.

4.3. Expressing network properties in PDDL

We formalise the encoding as PDDL proposed in section 3 for P1 and P2 as:

- $init = Pos_0 \wedge PKT_0$
- $goal = Pos_G \bigwedge_{(c,e)\in O_{|G}} \neg c \wedge VALID$

The definition of the goal is slightly more complicated than what expressed so far. In particular we need to be sure that the final destination of the packet is Pos_G and not only that the packet will go through Pos_G. We encode this condition by saying that the packet is in Pos_G and no other operator can be applied.

Note also that to evaluate $VALID$ we need to extend the actions defined previously in order to "record" the visit of a component; afterwards we can trivially express our goal as:

P5 As before I will present P5 first since P4 and P3 are simple sub-cases of P5.

To encode τ we will extend the planning problem into two problems connected by a sub-goal:

SETUP: Setup the initial PKT to satisfy τ,

SOLVE: Once the packet is fixed we solve the rest of the problem as we solved P2.

SOLVE will be a problem really similar to P2, except for the fact that all actions will be prefixed by a precondition *setupFinished*. And there will be an additional condition in the goal: *setupFinished*.

Similarly all actions in SETUP will have $\neg$*setupFinished* as precondition. SETUP will have k operators, where k is the number of bits in PKT (in our case 96), and each of them can turn one bit on.[6] There will be an additional operator checking whether the setup phase is over:

$$\langle (I \wedge \tau) \wedge \neg setupFinished, setupFinished \rangle$$

From this we encode P4 as P5 with an empty $VALID$ condition.

4.3.1. Verifying unreachability

As discussed in section 3.4.1, we must be careful when testing unreachability, since we need a complete planner: we need to show that, given a starting configuration we cannot reach the goal, i.e. there is no possible plan! As already mentioned, the termination test is usually impractical, and therefore we rarely obtain completeness. We could overcome this limitation by simulating what we are going to do with the SAT problem: add a bound. If we add a counter to our actions, we can check whether the counter reaches the bound without the goal being true; or we could use a planner that accepts an external bound as parameter (e.g. [24]). All these are feasible ways that would actually solve the problem, but we decided to continue at a lower level by considering the SAT encoding.

[6] This models the non-deterministic choice of an initial packet

5. Reconfiguration

Reconfiguration is the process of modifying the network component configuration in order to satisfy the Policy. We try to formally define the configuration in this section but intuitively, in the previous sections, we have been using the idea that the configuration is composed by the rules associated with each component. If we proceed this way, we quickly find out that each rule of each component can assume between 2^{70} and 2^{211} possible values (see appendix A.3). This result suggests that, to solve the problem, we need to restrict ourselves to "meaningful" configurations. How to do this is out of the scope of this work, nevertheless an overview of a possible solution is presented in appendix A.4.

We will present here a general solution in which we are given a finite set of possible configurations $\mathcal{C}$ and we have to choose one that satisfies the network policy, if it exists.

We start the section by an example before providing a formal definition of *configuration*. Afterwards, we study a first formalisation as QBF of both the generic problem and our simplification. Finally, we consider a few ways to reduce and solve the problem as SAT.

5.1. General idea

Lets start by introducing a small example on our sample network; we define 5 configurations by specifying a "Basic" configuration and stating the difference between this and the other configurations:

Basic The corrected version of the original configuration (defined in section 3), in which we swapped the conflicting firewall rules. This is one of the two acceptable configurations for our example policy.

Wrong Routing On R the router for 1.2 is defined as 2.2. This way the server becomes unreachable

Wrong Firewall As highlighted in section 3, the firewall rule (DstIP = 1.0/8, Reject) is placed before the rule (DstIP = 1.2/16, Accept). The Server becomes unreachable from the client network.

Block irrelevant port Adds a firewall rule blocking port 123. This is irrelevant to any of the policies defined in the example, and is the second acceptable configuration.

Allow Outside Traffic Allows access from 1.2 to the client network. This is against the policy.

In this work, we do not consider the possibility of topological changes to the network such as adding/removing hosts or changing addresses. Lets consider a firewall, we can see that its configuration is given by the ordered list of firewall rules. Therefore, when we define the configuration of a network, we simply define the ordered list of firewall/routing/NAT rules for a given set of hosts.

We can formalise this approach in two ways:

- Define a set containing all the rules for a single configuration, or

- Specify which rules apply (or not) to each component based on the configuration.

The two approaches differ significantly in the implementation: in the first case we build several "copies" of the network with different configurations, in the second case we have one copy of the network and we pick (on a rule by rule case) what to include.

We will analyse the second approach but we will see that the solution proposed works also in the first case.

5.1.1. Formalisation

Before being able to formalise the reconfiguration problem, we need to introduce some other definitions:

- Lets call $l = (Host, Type, Position, (Condition, Action))$ a configuration line. A configuration line contains the information about a rule, its type and its position for a particular host.
- let $\mathcal{L}$ be the set of all possible configuration lines,
- C_{ID} a set of identifiers for the possible configurations,
- we define $\mathcal{C} \subseteq C_{ID} \times \mathcal{L}$ as the relation linking to each configuration a set of configuration lines. We will, for simplicity, often use the functional notation $\mathcal{C}(c)$ to indicate the set $\{l | (c, l) \in \mathcal{C}\}$

With this definition, we cannot distinguish between the two methods described above; moreover, the two methods coincide if the configurations do not have any rule in common. Our preference for the second method is given by the fact that we expect configurations to share many configuration lines among each other.

In section 4 we defined the BRP with, among others, a network configuration C as parameter. We define $C = (c_{id}, topology)$ as *network configuration*, in which $c_{id} \in C_{ID}$ and *topology* is a function associating to each host a set of IP addresses.

Definition Given:

- A set of possible configurations $\mathcal{C} \subseteq C_{ID} \times \mathcal{L}$,
- a topology t,
- a network policy P

we call reconfiguration ($Reconf$) the problem of finding a network configuration $C = (c_{id}, t)$, such that in a network with topology t in which all the hosts are configured with all (and only) the rules from $\mathcal{C}(c_{id})$, the policy P is satisfied.

5.2. Solving the reconfiguration problem

5.2.1. Reconfiguration as QBF

We discussed in section 4 a way to reduce the BRP (and thus the network policies) to SAT. Therefore, to encode the reconfiguration problem, we simply extend this approach and use a quantified boolean formula (QBF):

$$R \approx \exists C.Policy(C) = \exists C.\phi_1(C) \wedge ... \wedge \phi_n(C) = \exists C.\forall Pos_0, PKT_0....$$

for all the properties ϕ_i in the network policy. If this QBF formula evaluates to true, then the configuration C is the one satisfying the policy. This formula should give an intuitive understanding of the encoding of the problem. We will analyse it in detail shortly. Before doing so, we would like to present a more generic encoding of the configuration problem, in which we are not provided with the set of configurations $\mathcal{C}$.

Lets associate to each configuration line in $\mathcal{L}$ an identifier. Now we are interested in knowing whether there is any subset of $\mathcal{L}$ that would satisfy our network policy. We express, intuitively, the problem as follows:

$$\exists l_0, .., l_m . \bigwedge_{i=0}^{k} \phi_i(l_0, .., l_m)$$

for all the properties ϕ_i in the network policy. With this encoding, we cover the complete "solution space" of the configurations but, as discussed before, we have a huge number of possible configuration lines, therefore this approach is intractable[1].

5.2.2. Redefining the component's encoding

We are now interested in how to properly encode the network components in order to include the information about configuration. This can be easily achieved by extending the precondition for each rule l by the configuration identifier. For a generic rule with precondition ψ_l we define the new precondition ψ'_l as:

$$(\bigvee_{c \text{ s.t. } (c,l) \in \mathcal{C}} c) \wedge \psi_l$$

Intuitively we *enable* the configuration lines that are part of the current configuration.

We can plug this extension in the encoding defined in section 3 without any additional modification, and define properly the reconfiguration formula R as:

$$R = \exists C.Policy \wedge C = \exists C.\phi_1 \wedge ... \wedge \phi_n \wedge C$$

with C being a suitable encoding of a single configuration identifier [2]. This formulation of the problem should make it clear that the encoding of the network policy is independent of the particular configuration being tested and, therefore, partially justifies the encoding as QBF.

5.2.3. QBF Solvers

Compared with SAT solvers, there are less QBF solvers publicly available, therefore to perform experimental evaluations, we focussed our search on solvers that took part on the QBFEVAL'10 competition ([1]) and in particular the one accepting QDIMACS formula as input.

Since we are providing the encoding of the verification problem through the Tseitin/definitional translation (section 6.2.1), we cannot solve directly the problem in the form $\exists\forall$ [3] Therefore, we want to prove that a given formula $\forall\exists$ is false and extract a counter example configuration from the universal quantification.

[1] We will show this approach to be Σ_2^p-complete.
[2] See section 3 for a discussion on how to encode numbers
[3] These translations preserve satisfiability of the formulae but not validity, that is what we are interested in.

An alternative encoding makes use of the positive formulation of the QBF formula, in which we explicitly quantify the Tseitin variables:

$$R \approx \exists C. \forall PKT, Pos. \exists aux. P*$$

We couldn't solve this formula with the QBF solvers[4] even for a single configuration. We think that the main reason for this behaviour is the universal quantification of almost 100 variables and the fact that QBF solvers are not as efficient as SAT solvers. Therefore, a solution purely based on QBF is an interesting problem to solve; we think that a good approach would be using non-clausal QBF solvers ([26], [13]).

5.3. Reduction to SAT

We present three interesting aspects of solving the reconfiguration problem using a SAT solver. The last one (incremental SAT solving) is the one we adopted for our implementation.

5.3.1. Σ_2^p as SAT

Propositional problems in the form $\exists x_1, .., x_n \forall y_1, .., y_n . \varphi$ are used to define the complexity level Σ_2^p of the polynomial hierarchy. If we add the assumption of φ being in CNF we obtain an interesting property: the problem can be reduced to SAT in linear time. The reduction defines a new problem φ' in which we simply remove all the variables that are universally quantified. Intuitively once we fixed a value for $x_1, .., x_n$ we want to test the validity of the formula CNF formula φ, i.e. we check whether $\varphi = T$ without considering $y_1, .., y_n$.

Unfortunately, since we compute *Policy* through the Tseitin conversion, we cannot apply this method: we only preserve satisfiability and not validity! Nevertheless, it would be interesting to study a method to generate our formula already as CNF and apply this technique.[5]

5.3.2. Shannon expansion

Since the number of configuration is limited we can reduce the 2QBF problem into a SAT problem, by abstracting the *existential* quantified ($\exists$), with the Shannon expansion, i.e.:

$$R = \exists \forall ... = \forall((Policy[C/\top]) \vee (Policy[C/\bot]))$$

This requires making a copy of *Policy*. As described later on (section 6) we only need to copy the transition relation (RT) that contains all the informations that are influenced by the configurations; unfortunately this is the biggest part of our encoding. Nevertheless, if we encode the configuration in binary, this operation leads to an increase of the formula size linear in the number of configurations.

At the beginning of the section we discussed two ways of encoding the network configuration. The Shannon expansion follows the first method, in which we create a copy of the network for each possible configuration.

This is an interesting approach to investigate, since it maintains the problem tractable.

[4] sKizzo [8] and quantor [9]

[5] Note that the same idea applies for Π_2^p in DNF.

5.3.3. Linear search

Since we want to find a single configuration that satisfies our policy, we can go through all of the configurations and test them individually. Recalling that we are dealing with the negation of our TAUT problem, for a given configuration C we test that $Policy \wedge C$ is UNSAT.

This method has complexity linear in the size of the possible configurations number: on average we will test half of the configurations before finding the good one. We adopted this method because it is really simple. Nevertheless, we were interested in understanding whether it was possible to improve the runtime.

Incremental SAT solving When solving a problem φ, a SAT solver performs a series of actions (pre-processing and clause learning) that are repeated when solving $\varphi \wedge \psi$ by a traditional solver. An incremental SAT solver "remembers" this information and is able to perform better for incremental problems.

If we encode our configurations as unary variables, we can perform our linear search by providing to the incremental SAT solver the following problems:

$$
\begin{aligned}
&Policy \wedge \\
&Policy \wedge \neg c_1 \\
&Policy \wedge \neg c_1 \wedge \neg c_2 \\
&...
\end{aligned}
$$

with $c_1, .., c_2$ being the configurations obtained as models from the SAT solver in the previous iteration. This is a negative search: we exclude one by one all the configurations that are not "good".

Another possibility is to simply test one by one the configurations. To do so we need a solver that accepts assumptions ([12]). This means that each time we change our assumptions (i.e., the configuration to test) the solver forgets the information related to the previous assumption and solves the new problem.

We expect both methods to perform better than running many times the SAT solver from scratch. In general we can say that the first method will evaluate all the *invalid* configurations before returning a valid one. The second will on average analyse half of the configurations. The worst case complexity is the same for both and is linear in the size of the configuration number.

In the first case, we get a counter-example each time. This might be useful to guide our search using some heuristic based on the reason why the previous configuration failed.

6. Implementation

In this section we explain the implementation choices done in NetSAT.

We start by introducing the basic idea behind bounded reductions to SAT (planning in particular) and present the concept of backward search. Later on we discuss some aspects related to the manipulation of big propositional formulas, their encoding as SAT and some improvements that we can perform over the SAT encoding.

6.1. Bounds in SAT

When performing a bounded SAT encoding, the general idea is to define a transition relation $R(i, i+1)$ and then write the transition relation $R(0, n)$ as $\bigwedge_{i=0}^{n} R(i, i+1)$. Therefore the final formula would look like:

$$\models I \wedge R(0,1) \wedge .. \wedge R(n-1,n) \rightarrow G$$

Since our goal is usually well defined, and we might be interested in checking different initial states over the same goal, we will present a different approach in which we build a formula that describes the set of initial states that can reach a particular goal.

6.1.1. Formalisation of the regression function

In our network problem, at most one action can be carried out at any time. This means we are looking for a sequential plan. We can rewrite each operator as an equivalence relation between variables at the current and the next state: $\phi_A \leftrightarrow \phi_{A'}$ [1].

To do so we introduce the relation $E : A^* \times \mathcal{O}$ defined from the set of literals $A^* = \{a, \neg a | a \in A\}$ to the set of operators $\mathcal{O}$ s.t.

$$E = \{(l, o) | o = (c, e) \in \mathcal{O} \wedge l \in e\}$$

The intuitive reading of E is to associate a literal l to the operators o that make it true.

We now define two sets of equivalences: $\mathcal{P}$ and $\mathcal{E}$.

$$\begin{aligned}\mathcal{P} =& \{o_i \leftrightarrow c | o_i = (c, e) \in \mathcal{O}\} \\ \mathcal{E} =& \{a' \leftrightarrow (a \bigwedge_{(\neg a, o^-) \in E} \neg o^-) \bigvee_{(a, o^+) \in E} o^+ | a \in A\}\end{aligned}$$

Intuitively $\mathcal{P}$ are the preconditions c for an operator o to be used, and $\mathcal{E}$ expresses that a variable is true if it was true and no operator changed it (o^-), or if an operator made it true (o^+).

We define the final set of equivalences $\mathcal{R}$ by combining $\mathcal{P}$ and $\mathcal{E}$, replacing every occurrence of o_i with the condition c_i s.t. $o_i \leftrightarrow c \in \mathcal{P}$:

[1] We use $'$ to indicate the next state

$$\{a' \leftrightarrow (a \bigwedge_{(\neg a,o^-)\in E, o^- \leftrightarrow c \in \mathcal{P}} \neg c) \bigvee_{(a,o^+)\in E, o^+ \leftrightarrow c \in \mathcal{P}} c | a \in A\}$$

$\mathcal{R}$ contains a set of equivalences expressing which conditions are needed for a variable to be true in the next state.

For simplicity we will use $\mathcal{R}$ as a function that substitutes to a proposition letter the equivalent formula. We call $\mathcal{R}$ the regression function.

Example We now introduce an example to clarify the definitions introduced. Note that this applies to all planning problems, but we will stick to the topic of networking. Lets take a network defined by the following operators:

$$\begin{aligned}
&\mathcal{O} = \{ \\
&o_1 = \quad \langle Pos_0, \neg Pos_0 \wedge Pos_1 \rangle, \\
&o_2 = \quad \langle Pos_1 \wedge A, \neg Pos_1 \wedge Pos_2 \rangle, \\
&o_3 = \quad \langle Pos_1 \wedge \neg A, \neg Pos_1 \wedge Pos_3 \rangle, \\
&o_4 = \quad \langle Pos_2, \neg Pos_2 \wedge Pos_4 \rangle, \\
&o_5 = \quad \langle Pos_3, \neg Pos_3 \wedge Pos_5 \rangle \}
\end{aligned}$$

We compute E as :

$$\begin{aligned}
E = \{ & \\
& (\neg Pos_0, o_1) \\
& (Pos_1, o_1), (\neg Pos_1, o_2), (\neg Pos_1, o_3) \\
& (Pos_2, o_2), (\neg Pos_2, o_4) \\
& (Pos_3, o_3), (\neg Pos_3, o_5) \\
& (Pos_4, o_4) \\
& (Pos_5, o_5) \}
\end{aligned}$$

and $\mathcal{P}$ as:

$$\begin{aligned}
\mathcal{P} = \{ & \\
& o_1 \quad \leftrightarrow Pos_0 \\
& o_2 \quad \leftrightarrow Pos_1 \wedge A \\
& o_3 \quad \leftrightarrow Pos_1 \wedge \neg A \\
& o_4 \quad \leftrightarrow Pos_2 \\
& o_5 \quad \leftrightarrow Pos_3 \}
\end{aligned}$$

Now we can compute $\mathcal{E}$:

$$\begin{aligned}
\mathcal{E} = \{ & \\
& Pos'_0 \quad \leftrightarrow (Pos_0 \wedge \neg o_1) \vee \bot \\
& Pos'_1 \quad \leftrightarrow (Pos_1 \wedge \neg o_2 \wedge \neg o_3) \vee o_1 \\
& Pos'_2 \quad \leftrightarrow (Pos_2 \wedge \neg o_4) \vee o_2 \\
& Pos'_3 \quad \leftrightarrow (Pos_3 \wedge \neg o_5) \vee o_3 \\
& Pos'_4 \quad \leftrightarrow (Pos_4 \wedge \neg \bot) \vee o_4 \\
& Pos'_5 \quad \leftrightarrow (Pos_5 \wedge \neg \bot) \vee o_5 \\
& A' \qquad \leftrightarrow (A \wedge \neg \bot) \vee \bot = A' \leftrightarrow A \}
\end{aligned}$$

Finally we compute $\mathcal{R}$:

$$
\begin{aligned}
\mathcal{R} = \{ & \\
& Pos'_0 \quad \leftrightarrow (Pos_0 \wedge \neg Pos_0) = \bot \\
& Pos'_1 \quad \leftrightarrow (Pos_1 \wedge \neg(Pos_1 \wedge A) \wedge \neg(Pos_1 \wedge \neg A)) \vee Pos_0 = Pos_0 \\
& Pos'_2 \quad \leftrightarrow (Pos_2 \wedge \neg Pos_2) \vee (Pos_1 \wedge A) = (Pos_1 \wedge A) \\
& Pos'_3 \quad \leftrightarrow (Pos_3 \wedge \neg Pos_3) \vee (Pos_1 \wedge \neg A) = (Pos_1 \wedge \neg A) \\
& Pos'_4 \quad \leftrightarrow (Pos_4) \vee Pos_2 \\
& Pos'_5 \quad \leftrightarrow (Pos_5) \vee Pos_3 \\
& A' \quad \leftrightarrow A\}
\end{aligned}
$$

$\mathcal{R}$ shows us how to make a particular propositional variable true: e.g. we see that there is no way to make Pos_0 true in the next step, and that A never changes its value.

6.1.2. Backward Search

The purpose of building $\mathcal{R}$ was to be able to perform a backward search: given the goal we want to build the set of initial conditions that solve the problem. We do so by applying the function $\mathcal{R}$ iteratively to each propositional variable until the formula satisfies our initial state.

As an example take the goal condition $G = Pos_4$ and the initial condition $I = Pos_0 \wedge A$:

$$
\begin{aligned}
G_0 = \quad & Pos_4 \\
G_1 = \quad & Pos'_4 = Pos_4 \vee Pos_2 \\
G_2 = \quad & Pos''_4 = Pos'_4 \vee Pos'_2 = (Pos_4 \vee Pos_2) \vee (Pos_1 \wedge A) \\
G_3 = \quad & G'_2 = (Pos'_4 \vee Pos'_2) \vee (Pos'_1 \wedge A') = \\
& = ((Pos_4 \vee Pos_2) \vee (Pos_1 \wedge A)) \vee (Pos_0 \wedge A)
\end{aligned}
$$

$$I \wedge \chi \models G_3$$

where χ expresses the mutual exclusivity of the Pos propositions [2]

One of the key features of this process is that a variable stays the same unless something changes it. Therefore we can compute G_n that encodes the goal for paths of "at most" size n. We call such formula *regression*.

After computing the regression of G_n we check whether the starting condition belongs to the goal, i.e. if $\exists.I \wedge \chi \wedge \neg G_n$ is true.

Note that the more the goal is specified, more substitutions will be performed during the regression steps. Nevertheless many of this substitutions will lead to contradicting/tautological sub-formulae. This justifies spending some time trying to simplify the regression.

The reason why we are interested in this is to be able to answer multiple queries regarding the same goal. In fact, once we compute a regression for a given goal (and an n big enough) we can store it and reuse it to solve all the queries that have the same goal.

6.2. Formulae as Circuit

Performing regressions can make the formula grow exponentially, therefore we are interested in keeping the representation of the formula as compact as possible.

In general a propositional formula can be represented as a tree with each leaf node being a proposition.

A slight improvement in compactness can be achieved by reusing the same propositions, this way we go from a tree to a DAG. By applying the same idea to sub-formulae, we avoid having

[2] i.e., $Pos_i \leftrightarrow \bigwedge_{j=0}^{5} \neg Pos_j$: at most one position can be true at any given time

repetitions of the same sub-formulae. The main advantage of this approach is not only the smaller amount of memory that we use, but also the fact that computing the Tseitin/Definitional CNF can be performed efficiently by simply traversing the DAG.

There are different kinds of circuital representations of boolean functions (mostly derived from electronic CAD):

AIG : AND-Inverter Graph, is a DAG that only uses AND gates and Inverters. The inverters are usually expressed on the line/edge, so in practice there's only one type of gate/node ([17]).

NICE DAGs : Negation, Ite, Conjunction, Equivalence are the only gates available. This kind of DAG was designed to perform better conversion to CNF to be fed to SAT solvers. This is implemented also in a tool called BAT ([19]).

RBC : Reduced Binary Circuits are similar to AIG but allow binary gates only (and equivalence). Even though is quite plausible that by using a functionally complete set of operators one can get a maximal sharing, this can increase the depth of the dag in case of big conjunction/disjunctions ([3]).

We were not able to find a proper comparison among this methods. Therefore, to maintain our implementation simple, we decided to use n-ary gates and allow the explicit use of AND,OR and NOT. We are confident that a better circuital encoding can probably speed up significantly our implementation.

6.2.1. Tseitin CNF

Given a propositional formula we can build an equisat CNF formula by applying the Tseitin method ([28], also known as definitional translation). The idea is to assign a brand-new propositional variable to each sub-formula and build a new conjunctive formula where each clause represents the equivalence between the propositional variable and the sub-formula: e.g.

$$
\begin{array}{rcl}
 & (A \vee (d \wedge e) \vee (b \wedge c) \vee (a \wedge (b \vee (d \wedge e)))) & = \\
= & (A \vee EQ1 \vee (b \wedge c) \vee (a \wedge (b \vee EQ1))) \wedge (EQ1 \leftrightarrow (d \wedge e)) & = \\
= & (A \vee EQ1 \vee (b \wedge c) \vee (a \wedge EQ2)) \wedge (EQ1 \leftrightarrow (d \wedge e)) \wedge (EQ2 \leftrightarrow (b \vee EQ1)) & = \\
= & (A \vee EQ1 \vee (b \wedge c) \vee EQ3) \wedge & \\
 & (EQ1 \leftrightarrow (d \wedge e)) \wedge (EQ2 \leftrightarrow (b \vee EQ1)) \wedge (EQ3 \leftrightarrow (a \wedge EQ2)) & = \\
= & (A \vee EQ1 \vee EQ4 \vee EQ3) \wedge & \\
 & (EQ1 \leftrightarrow (d \wedge e)) \wedge (EQ2 \leftrightarrow (b \vee EQ1)) \wedge (EQ3 \leftrightarrow (a \wedge EQ2)) \wedge (EQ4 \leftrightarrow (b \wedge c)) &
\end{array}
$$

The resulting formula is (almost) in CNF. Note how this process can be enhanced if we use a circuital representation: in the first step we substitute two instances of the same sub-formula with only one propositional variable in one step. To complete the conversion to CNF we only need to "expand" the equalities.

It should be clear by now that circuital representation and Tseitin encoding are crucial to be able to operate fast on big formulas: the computation of the Tseitin CNF is performed by visiting the DAG thus linear in the size of the circuit.

6.3. Using regression as transition relation

The advantage of using the regression from the goal is evident if we have many queries over the same goal. In general we might not have enough queries to justify the pre-computation of

the regression. Therefore we decided to use the regression as a transition relation: we compute $R(n, n+1)$ and then perform many copies of the circuit. The main difference between this method and the transition relation (as defined in [10]), is that we are able to analyse solution of size "at most" n instead of having to choose a priori the bound. The benefit of this will become evident later, when discussing how to merge multiple queries in a single SAT instance.

The creation of the relation $R(0, n)$, being it computed forwards or backwards, can be sped up by computing it at the CNF level. In particular what we did was:

1. Compute a single regression step for each propositional variable and build the circuit representing $R(n, n+1)$
2. Extract the CNF from the circuit and convert it to DIMACS
3. Duplicate the CNF/DIMACS formula n times, shifting the variables each time
4. Add a set of equivalence clauses "linking" the variables from the different steps.

In our case this is possible because the regression equivalences only depend on the configuration and not the query.

6.4. Multiple queries in a single SAT problem

Recall that to verify any property, we need to check whether the encoding is unsatisfiable. Moreover, the regression table RT is independent from the queries: what we change are the initial state (I,τ) and the goal (G,$VALID$).

Thus if we have k queries (Q_0,...,Q_k) we can compute the regression table RT and write the UNSAT problem:

$$\exists.RT \wedge (Q_0 \vee ... \vee Q_c)$$

with, e.g. $Q_0 = \neg\phi_{BRP} = (I \wedge \tau) \wedge \neg(G \wedge VALID)$. This way, we avoid having k calls to the sat solver, by trying to satisfy all the properties at the same time. This encoding is important to be able to encode the reconfiguration problem compactly, since we are interested in a configuration that satisfies all the properties at the same time.

The only disadvantage with this approach is the precomputation of RT for a fixed bound n; this force us to use a bound that is "big enough" to satisfy all the properties, but it is usually the case that reachability properties can be solved with a small bound k, while to be sure of an unreachability property we need an higher upper-bound.

7. Complexity analysis

This section studies the complexity of the basic reachability problem (section 4) and the reconfiguration problem (section 5).

We will define two different complexities:

Network The complexity on the number of nodes in the network with a packet of fixed size

Packet The complexity on the size of the packet in a network with a fixed number of components

Before justifying this choice, lets roughly summarise the complexity results for our problems[1]:

	Network	Packet
BRP	P	coNP-complete
Reconf	NP-complete	Σ_p^2-complete

In general, we are much more interested in network rather than packet complexity. The idea is that we are likely to work with networks of different dimensions but keep the packet fixed in size. On the other hand, we can see how the complexity of the problem is mainly given by the packet size therefore, any extension of the framework to higher or lower protocols will have to deal with this complexity increase.

These results also justify the reductions to QBF and SAT that we are using to solve the problems.

We will now present the proofs for these complexity results by first studying the network and later the packet complexity. Later on, we discuss how these results are influenced by the decision of picking a fixed constant k as upper bound for our BRP[2]. We conclude the section by reviewing the results in the framework of parameterized complexity ([11]).

7.1. Network Complexity

7.1.1. Basic Reachability Problem

When defining the Basic Reachability Problem, we need to specify the following parameters:

1. a network configuration C
2. an initial position Pos_0,
3. a formula characterising a non-empty set of initial packets τ,
4. a formula characterising the path $VALID$
5. a final position Pos_G,

[1] We prove hardness for the most expressive version of the reconfiguration problem, in which we are not provided with a set of configurations.

[2] We do not include the results for the extended reachability problem (BRP^*), in which we quantify over the initial position; the complexity of this problem coincides with the complexity of the BRP, in fact we simply solve the BRP for each initial position (that are a constant).

6. and an integer n as upper-bound.

Since the initial and final position denote a particular network component, they are not influenced by the size of the network; τ being a finite set of initial packets[3] depends on the size of the packet but not on the size of the network. Thus the network configuration, $VALID$ and the upper-bound are the only parameters that change. In fact C explicitly states which are the components of the network, while $VALID$, being defined over the visited nodes, might contain all the nodes of the network. Finally the upper-bound is a particular case, since we defined a value for it that depends on the network size (mp) and another value that is a constant (i.e. n=1300) (see 7.3); since we want to do an asymptotic analysis we will consider the upper-bound to be equal to the constant.

Theorem 1. *When the packet size is fixed (network complexity) BRP is in P.*

Proof. To show this we will present a polynomial time algorithm that can solve the problem.

Algorithm 1 BRP Algorithm

```
for packet ∈ τ do
  Pos_curr ← Pos_0
  while ¬GoalReached ∧ bound ≠ upper − bound do
    Visited_curr ← true
    for R ∈ RuleTable do
      if packet ⊨ R then
        if Pos*_n == Pos_curr == Pos_G then
          GoalReached ← True
          break
        end if
        Pos_curr ← Pos*_n
        bound++
        break
      end if
    end for
  end while
  if bound == upper − bound ∨ VALID ⊨ ⊥ then
    return false
  end if
end for
return true
```

The outer loop is over a constant (the number of initial packets) while the innermost loop is on the rules. Each table has a finite size that is independent from the number of network components, and has a maximum size determined by the size of the packet[4]. Since evaluating $VALID$ can be done in polynomial time, the overall complexity is polynomial. □

Lemma 1. *The network complexity of P1 and P3 is linear.*

[3] τ is not a set but a formula characterising a set of initial packets; nevertheless, since there's a constant number of packets, we can build such set in linear time.

[4] NAT Rules are the most complex, since they have a precondition on $|PKT|$ bits and an action on $|PKT|$ bits. Thus there are $2^{(2|PKT|)}$ possible NAT Rules

Proof. The only polynomial step in algorithm 1 is checking the evaluation of $VALID$, since $VALID = \top$ for both $P1$ and $P3$, we can solve them in linear time. □

As a side remark, note that we do not require $VALID$ to be in any particular form. If we assume it to be in CNF, then the evaluation of $VALID$ could be performed in linear time and also P2 would have linear complexity.

7.1.2. Reconfiguration

In the reconfiguration problem presented in section 5 we assumed that a set of possible configurations was provided. This is a simplification we had to introduce for tractability, but it is not really interesting from the complexity point of view [5]. Therefore here we consider the original configuration problem, in which each component can have several configurations and we need to choose a good configuration for each of them.

Since we know that the possible configurations for a single component are independent from the size of the network but different components in the path might interact in different ways, we obtain:

Theorem 2. *Reconf is in NP in network complexity.*

Proof. Lets fix $conf$ as the maximum number of possible configurations for a single component; we know that in a path composed by n components we need to check the BRP for $conf^n$ possible network configurations.

Nevertheless since we are interested in only one configuration, we can make a non-deterministic guess of the configuration of each component ($conf * n$ bits, thus linear in the size of the network) and then verify the BRP. Adding the non-deterministic guess to the results of Theorem 1 gives us the NP complexity. □

We prove NP-hardness (thus completeness) of Reconf by reduction from SAT.

Given a CNF formula ϕ we build the reconfiguration problem of the network $\mathcal{N}$ and we state the following lemma:

Lemma 2. *A CNF formula ϕ with n propositional variables is satisfiable iff the Reconf problem of the network $\mathcal{N}$ has a solution.*

Proof. Given ϕ in CNF we build $\mathcal{N}$ as follows:

- Define an initial ($START$) and final (END) host
- For each propositional variable x in ϕ create 2 routers R_x and $\overline{R_x}$
- Define $VALID_\phi$ by replacing in ϕ the occurrences of x with $Visited_{R_x}$.
- Connect the routers in the following way:
  - We define an order for the variables (i.e., $x_1, .., x_n$) and we connect $\overline{R_{x_i}}$ to $\overline{R_{x_{i+1}}}$
  - We additionally connect each $\overline{R_x}$ router to the corresponding R_x router; the $START$ host will be connected to the first router and the last $\overline{R}$ and R routers will be connected to END
  - Finally we connect R_{x_i} to $\overline{R_{x_{i+1}}}$
- We define two possible configurations for each $\overline{R_{x_i}}$ router: in one case it will forward the test packet to the R_{x_i} router in the other case it will forward it to the $\overline{R_{x_{i+1}}}$ router.

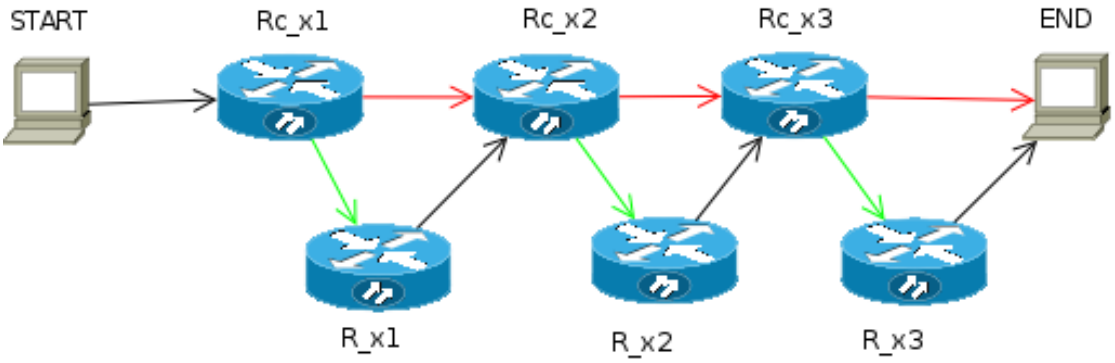

Figure 7.1.: Reduction Example

Figure 7.1 shows an example of the network $\mathcal{N}$. Intuitively the configuration of the $\overline{R_x}$ router says whether the R_x router will be visited, thus whether the variable $Visited_{R_x}$ would be set to true or false.

We define only one network property for the policy, with the newly defined $VALID_\phi$, initial position $START$, goal position END and upper-bound $2 * n$.

The reconfiguration problem is to pick among the two possible values of the configuration of each $\overline{R_{x_i}}$ in order to satisfy the policy.

In particular the relation between the configuration of $\mathcal{N}$ and any interpretation I is as follows:

$$\begin{cases} \overline{R_{x_i}} \Rightarrow R_{x_i} & \text{iff } x_i^I = \top \\ \overline{R_{x_i}} \Rightarrow \overline{R_{x_{i+1}}} & \text{iff } x_i^I = \bot \end{cases}$$

Since $VALID_\varphi$ is equisatisfiable to φ, we just need to show that $VISITED_{R_x} \equiv x$ for every x, but this simply follows from the relation between I and $\overline{R_{x_i}}$. □

From Theorem 2 and Lemma 2, it follows that:

Theorem 3. *Reconf is NP-complete in network complexity.*

7.2. Packet complexity

We introduce the packet in section 3 as composed by some of the headers of the TCP/IP protocol. Therefore, by packet complexity we mean the complexity related to the size of this headers, on which each component can act. This analysis is useful when considering different protocols from the ones covered in this report: e.g. the transition from IPv4 to IPv6, that would make our packet grow from 96 to 288 bits.

7.2.1. BRP

Theorem 4. *When the network size is fixed (packet complexity), P1 can be decided in linear time.*

[5] We need to make some additional assumptions on the relation between the configuration set and the network size that are hard to justify and, in general, we are not able to prove the hardness result.

Proof. This result is justified by the fact that we only need to run algorithm 1 for one particular packet and that we can store the rules in order to have a linear lookup time. □

This nice complexity does not carry on with more expressive properties. In particular, the number of times that we need to test the BRP increases exponentially (in the worst case) with the size of the packet. To avoid having to enumerate this exponentially many packets, we decided to encode the initial packets set with the propositional formula τ. The proof for completeness presented here works if we remove the assumption of τ being non-empty. We can safely remove this assumption without changing anything else; the reason why we include this in our definition is that otherwise any property would be trivially satisfied.

We prove therefore the following result:

Theorem 5. *BRP is coNP-complete in packet complexity.*

Lemma 3. *$BRP \in$ coNP in packet complexity.*

Proof. From Theorem 4 we see that the complexity is not given when analysing a single packet, but from the fact that we have an exponential number of packets to check. We can verify the BRP by reducing it to a non-deterministic guess: we can guess a packet that belongs to the set τ but does **not** satisfy the property. Choosing a packet requires a number of guesses that is linear in the size of the packet, and verifying whether it satisfies τ can be done in polynomial time. Finally, we run the algorithm and eventually check whether the run satisfies $VALID$. Since both steps are polynomial, we can say that the complexity of this operation is NP. Finally, to prove the BRP we need to show that such a packet *does not* exist, therefore we deal with the complementary problem that is, by definition, in coNP. □

We prove hardness with the following reduction from UNSAT:

Lemma 4. *Given a formula ϕ the BRP problem P is satisfied iff ϕ is UNSAT.*

Proof. Lets define ϕ_{PKT} as a rewriting ϕ in which we replace each propositional variable in ϕ with a packet bit: if ϕ has n variables, we will consider a packet of length n and associate to each variable one of those bits. We build a network composed by two hosts (Start and End) connected by a firewall that will have only one rule that denies all traffic. We finally define the network property P with $\tau = \phi_{PKT}$, initial position Start, final position End and $VALID = \top$. Intuitively, we are asking whether all the packets from Start can reach End. This is true if and only if τ is empty, that is equivalent to say that ϕ is UNSAT. □

7.2.2. Reconf

If we consider a finite set of available configurations, we simply have a constant number of configurations to try, therefore the complexity is the same as the BRP.

An interesting question is what happens if we predefine a relation between the configurations and the behaviour of the component. Note that this relation must be part of the original problem, because otherwise it would take exponential space to be written and read as input.

We present now a particular version of $Reconf$, that we will call $Reconf^*$, in which we allow k rules for each component. Each of the k rules will be associated with a number of configuration variables depending on the type of the rule. E.g. For a firewall F, we will define:

- $cR_{F,1}, ..., cR_{F,j}, ..., cR_{F,k}$ to indicate whether the j-th rule is active or not
- we encode the $2^{|PKT|}$ possible conditions on the packet and the 2 actions as a binary number $c_{F,1,1}, ..., c_{F,1,|PKT|+2}$. We will encode the source IP address, for example, always in the first 32 variables etc.

Note that the number of configurations depends on the type of component, for example for NAT we have $2^{2*|PKT|}$ possible configurations[6]. Finally, we extend the alphabet of $VALID$ also to the configuration variables, therefore providing a way to enforce some "necessary" configuration among all the available configurations.

This extension of the problem allows us to state the following result:

Theorem 6. *$Reconf^*$ is Σ_2^P-complete in packet complexity.*

Proof. We can show that $Reconf^*$ can be solved by reduction to QBF similarly as for $Reconf$ (section 5). The only additional constraint we must add is to pick a constant k representing the maximum number of rules per component, and encode the configurations in the problem and not as input in order to maintain the size of the problem polynomial. We prove hardness (thus completeness) in Lemma 5. □

Lemma 5. *A given 2QBF formula $\exists\forall\varphi$ is satisfiable iff the $Reconf^*$ problem P has a solution.*

Proof. Given the 2QBF formula of the type $\exists.x_1, .., x_n \forall.y_1, ..., y_m.\varphi$, we build the $Reconf^*$ problem P as follows:

- We take a network with a Start and End host and we connect them through a NAT N
- We define the packet size as $|PKT| = n + m$
- We associate to x_i the configuration variable $c_{N,1,i}$ such that this variable defines the following behaviour: $c_{N,1,i} = \top$ iff $PKT[i] = \top$, where with $PKT[i]$ we mean the action of the NAT modifying the i-th bit. Intuitively, we set the value of the first n bits of the packet to correspond to the value of the n existentially quantified variables.
- We define $VALID_\varphi$ in terms of the packet bits from φ, in particular we replace x_i in φ with $PKT[i]$ in $VALID_\varphi$ and the variables y_j with $PKT[n + j]$. Therefore our formula $VALID_\varphi$ is equivalent (up to variable renaming) to φ.
- Since in our new problem we have many configurations variable, we set them all in such a way that the only active rule is the one defined above. Similarly we set an empty configuration for the hosts Start and End. To do so, we simply extend $VALID_\varphi$ with the relevant equivalences (e.g. $cR_{N,1} \leftrightarrow \top$). In this step we lose equivalence between $VALID_\varphi$ and φ but we preserve equisatisfiability.
- Finally we define the Reconf problem as usual with Start as initial position, End as goal position, $VALID = VALID_\varphi$ and $\tau = \top$.

In simple words, what we did is simply to rewrite φ in terms of the packet variables, and configure the NAT in order to copy the value of the existentially quantified variables of the original problem in the packet. The reconfiguration problem has a solution iff there exists a configuration such that all packets can reach End from Start (trivially true) and satisfy $VALID_\varphi$ that is equivalent to say that φ is satisfiable. □

[6]For an explanation of this number see appendix A.3

7.3. Unbounded analysis

We consider now the importance of having a fixed upper-bound for the BRP problem.

For what concerns the network complexity, if we didn't have an upper-bound, there would be no difference in terms of worst case complexity. Nevertheless, the natural upper-bound of the problem is given by $|Host|^{2^{|PKT|}}$ meaning that we can detect a loop only after a huge number of steps. This becomes relevant when considering packet complexity since the upper-bound depends as a double exponential on the size of the packet. These results suggest that it is necessary to use some bound that can be deduced from the domain. We did so by introducing two possible values.

7.3.1. TTL in IP packets

One of the fields of the IP packet is called Time-To-Live (TTL). The goal of this field is to avoid packets from travelling without end in the network; each time a router processes a packet, it decreases the value of the TTL. When this value reaches 0, the router drops the packet. Since we are looking for a way to detect loops, we can simply reuse this existing solution by modifying it slightly to fit our model. In the IP packet the TTL field can assume up to 256 values ([2]). Since in our model we expand each host into many component, we need to take this into account. We define this upper-bound as $mp = 256 * 5 \approx 1300$, with 5 being the maximum number of components between two hosts in our expanded network.

7.3.2. Maximum Path (mp)

We know that each packet can go through a node only once. The problem lies in how NAT modifies the packet. But we know that there's only a finite (and hopefully small) number of modifications that can be applied at each time by a NAT. We can then define the maximum number of modifications a packet can have. To do so we recall that a packet can go through a node only if this node supports forwarding. Lets then define FR as the number of forwarding hosts in the network and $|NATRules|$ the total number of NATRules in the network. $mp' = (FR*|NATRules|+1)$ (where the $+1$ stands for the the goal Host). This describes the maximum number of hosts that can be crossed before the packet reaches the destination. The worst case scenario is having two routers that forward to each other the packet each time applying one of the NATRules. Since each host is connected to the other through a subnet, we need to consider also $NATRules + 1$ subnets as intermediate steps. Moreover since we expand each host into up-to 5 components the correct formula is: $mp = 5(FR * |NATRules| + 1) + |NATRules| + 1$.

In general mp has small values for simple networks where the number of routers and NAT rules is limited. Therefore when picking the upper-bound is always a good idea to choose the minimum between mp and 1300.

7.4. Parameterized complexity

The analysis we propose in this section is somehow unorthodox, in the sense that we do not consider the worst-case running time in relation to the entire input, but we make a difference between network and packet size.

A more rigorous approach to a similar concept is developed in the framework of *parameterized complexity* ([11]), in which we consider decisions problems as composed by an input and a parameter. Given a decision problem and a parameter $k \in \mathbb{N}$, we say that the problem is *fixed-parameter tractable* (FPT) if it can be solved in $\mathcal{O}(f(k) * n^c)$, with n size of the problem, for

some constant c and computable function f. The name FPT derives from the idea that, once we fixed our parameter k, the rest of the problem can be solved in polynomial time.

7.4.1. BRP

In the context of parameterized complexity, we can rephrase Theorem 1 as follows:

Lemma 6. *BRP parameterized by the packet size is FPT.*

This follows from our proof of Theorem 1, since the number of initial packets is bounded by 2^k (with k being the packet size), we obtain $\mathcal{O}(2^k * n^c)$ (for some c) as worst-case running time, that matches exactly our definition of FPT.

This characterization of the problem is nice, because it allows us to identify the network complexity and the packet complexity in the same result: the packet size is related to k and the network complexity is related to n.

7.4.2. *Reconf**

We will now show that a parameterized complexity characterization is also possible for *Reconf** but, under the theoretical assumption that $FPT \neq W[2]$, this problem is not FPT.

We first recall that the weight of an interpretation corresponds to the number of variables that are mapped to true: e.g. $I = \{x, \neg y, z\}$ has weight 2. Weighted-SAT is the parameterized problem that asks whether a formula has a satisfying assignment of weight *exactly* k, for a given parameter k. This problem is complete for the corresponding class W[SAT]. If we restrict syntactically the input formula, we obtain different classes, for example Weighted-CNF SAT is the weighted SAT problem in which F is in CNF. Weighted-CNF SAT is W[2]-complete.

We can combine Theorem 2 and Theorem 6 in the parameterized complexity framework as follows:

Lemma 7. *Reconf** *parameterized by the packet size is W[2]-hard.*

Proof. As discussed in the proof of Theorem 6, the size of the packet influences the possible configurations lines of each component and, therefore, also the total number of possible configurations. This means that we can express any possible configuration of our component by deciding $f(|PKT|)$ bits (for some f that states the relation between the type of component and the possible configuration lines). Therefore, the natural parameter for *Reconf** is the size of the packet.

In the proof of Theorem 2, we use the idea of associating each variable x of a given formula F with a router R, and we check satisfiability by considering the rewriting of F in terms of $Visited_{R_x}$. We can reuse the same reduction to show W[2]-hardness, by simply adding two constraints that will guarantee that the solution has weight k for a packet size $g(k)$. These constraints force the number of visited routers R_{x_i} to be exactly k.

- For a weight k, we consider a packet of size $log(k) + 1$ that we will use as a counter (*cnt*);
- After each router R_{x_i} we add a NAT, with $k + 1$ rules that increment *cnt*: e.g. for $k = 4$ 000 $\Rightarrow$ 001,001 $\Rightarrow$ 010, ..., 011 $\Rightarrow$ 100;
- We add a firewall before END that drops all the packets for which $cnt \neq k$;
- Our test packet from $START$ will have $cnt = 0$;

Since in the reconfiguration problem we can reconfigure any component, we might wonder whether we can reconfigure the firewall in order to accept packets with $cnt < k$. To workaround this issue, we use the same constraint that we use in the proof of Lemma 5: we add in $VALID$ the relevant equivalences that enforce the rules stated above for all the components. However, we easily see that in order to generate a property $VALID$ with this constraints, we need to add $f(|PKT|) * n$ constraints to our problem (where f is exponential). Therefore, this is an fpt-reduction ([11]) since the time required is exponential only in the parameter k.

It should be clear now that the given formula F has a solution of weight k iff the network built with the previous reduction has a reconfiguration that makes the test packet from $START$ reach the destination END and satisfies $VALID$. □

7.4.3. Final remark

The study of the parameterized complexity of a problem is particularly interesting when we manage to prove membership in FPT, since this usually gives us an insight on how to solve efficiently our problem.

We saw that $Reconf^*$ is W[2]-hard, but its completeness is an open question. Nevertheless, completeness results for the W-hierarchy are mainly useful as refutation of FPT hardness, since they do not provide us with any idea on how to better solve the problem.

For practical applications of the $Reconf$ and BRP problem, we will probably fix the size of the the packet once, and then apply our algorithms for different networks of different sizes. This somehow justifies our interest in studying the complexity by considering only the packet size as parameter.

8. Conclusions

We conclude this report by showing an example run of our prototype NetSAT and discussing possible future directions of this work.

8.1. Experimental example

The prototype NetSAT [1] is a Java implementation of the work presented in this report, that relies on SAT4J for solving the SAT instances.

We present now the output of the program, when executed on our example network, with the policy and configurations described in section 4 and 5 respectively.

```
0 - C_wrongFw
1 - C_wrongRouting
2 - C_basic
3 - C_blockUselessPort

Expanded model Elapsed:122ms

[..]

R:

 IPs:
   192.168.0.1/24
   192.168.1.1/24
   192.168.2.1/24
 Routing Table:
   192.168.2.0/24   0.0.0.0
   192.168.1.2/32   192.168.2.2
   192.168.1.0/24   0.0.0.0
   192.168.0.0/24   0.0.0.0
   0.0.0.0/0   192.168.2.2
 Firewall Table:
   DestinationPort == 123  -> REJECT (C_blockUselessPort,)
   SourceIP == 192.168.0.0/24 DestinationIP == 192.168.0.1/32
       DestinationPort == 22  -> ACCEPT ()
   SourceIP == 192.168.0.0/24 DestinationIP == 192.168.0.1/32  -> REJECT ()
   SourceIP == 192.168.0.0/24 DestinationIP == 192.168.1.1/32  -> REJECT ()
   SourceIP == 192.168.0.0/24 DestinationIP == 192.168.2.1/32  -> REJECT ()
   DestinationIP == 192.168.1.2/32 DestinationPort == 80  ->
       ACCEPT (C_basic,C_wrongRouting,C_blockUselessPort,)
```

[1] Available at http://marco.gario.org/work/NetSAT/

```
   SourceIP == 192.168.1.2/32 SourcePort == 80  -> ACCEPT ()
   SourceIP == 192.168.0.0/24 DestinationIP == 192.168.1.0/24  -> REJECT ()
   SourceIP == 192.168.0.0/24 DestinationIP == 192.168.2.0/24  -> REJECT ()
   DestinationIP == 192.168.1.2/32 DestinationPort == 80  -> ACCEPT (C_wrongFw,)
   SourceIP == 192.168.0.0/24  -> ACCEPT ()
   SourceIP == 192.168.2.2/32 DestinationIP == 192.168.0.0/24  -> ACCEPT ()
    -> REJECT ()
 NAT Table:
   DestinationIP == 192.168.2.1/32 DestinationPort == 80  ->
         DestinationIP == 192.168.1.2/32
   SourceIP == 192.168.0.0/24 DestinationIP == 192.168.1.0/24  ->
         SourceIP == 192.168.2.1/32
```

We can see at the beginning the available configurations and the time required to build the expanded network. Afterwards, the configuration of each component is visualized (here reduced for brevity) with each line associated with the configurations it belongs to.

```
MP: 39

Equalities: 157

Regression:  Elapsed:1750ms
Problem Building Elapsed:373ms
Initial SAT Elapsed:1446ms
```

The parameter *mp* is visualized and used to build the regression table on (e.g.) 157 variables. We run a first instance of the SAT solver to verify if the reconfiguration problem is unsatisfiable, in the sense that none of the configuration is a solution. Note the running time ($\approx$ 1400ms), because in the next snippet we run the incremental SAT solver and the running time is almost one order of magnitude smaller.

```
Trying configuration: C_wrongFw
Solution unsat Elapsed:231ms
Property 3 is false
Counter Example Model:
DestinationIP == 192.168.1.2
DestinationPort == 80
SourceIP == 192.168.0.2
SourcePort == 6272
Pos == C1

Trying configuration: C_wrongRouting
Solution unsat Elapsed:74ms
Property 3 is false
Counter Example Model:
DestinationIP == 192.168.1.2
DestinationPort == 80
SourceIP == 192.168.0.3
SourcePort == 57455
Pos == C2
```

```
Trying configuration: C_basic
OK
Solution sat Elapsed:125ms
EX Elapsed: 8573
```

Finally, for each configuration that is not a solution, we get one of the properties that is not satisfied with a counter-example.

8.2. Open issues

We conclude this report by briefly looking into interesting possible developments of this work.

8.2.1. Reconfiguration as CTL

We could express the reconfiguration problem as a CTL ([7]) satisfiability problem. CTL allow us to alternate universal with existential quantification. We would, therefore, ask whether there is a state from which all the properties are satisfied. We would test this property on a model constructed by "reusing" the operators of the planning encoding as transition relations; this model would have a few initial states dedicated to *guessing* the configuration and, from there, we would check that for all the properties (encoded as different starting states for the chosen configuration), for all the initial packets, we can reach our goal for that property. Roughly we would work on a formula like

$$\exists \bigcirc\forall \bigcirc\forall\Diamond((p_1 \rightarrow G_1 \wedge VALID_1) \wedge ...(p_n \rightarrow G_n \wedge VALID_n))$$

and work on the model to define the possible configurations, the initial states of the properties and so on.

8.2.2. Improving the encodings

The encoding we provided for the different components can probably be improved. The first direction is to find a way to use preconditions that are "simple". The idea is that, when encoding ordered rules, we encode the negation of all the previous preconditions: the resulting formula is likely to contain parts that are trivially unsatisfiable or valid, and therefore we should be able to simplify it. A more detailed introduction to the problem is presented in appendix A.2.

The second direction is to allow a more expressive reconfiguration problem, by restricting the configuration lines to only the "meaningful" ones, i.e. addresses and ports related with the current policy and configuration. An introduction to this idea is presented in appendix A.4.

8.2.3. Extensions

By using the same idea of "decomposing" hosts in smaller components, it should be simple to consider services running on a particular address/port on a host and model more complex interactions between client and server. This could cover different application protocols (like http) by simply increasing the information stored on the packet.

The same ideas could be applied in the opposite direction, encoding lower levels. This would allow us to take into account full directionality and provide a more realistic model for our network.

One of the main challenges when dealing with this extensions are the complexity results (section 7), that show that the packet complexity might make the problem quickly hard to solve.

8.3. Final Remarks

In this work we introduced a rigorous formalization for two interesting problems related to computer networks: verification and reconfiguration. After introducing some assumptions we presented both a propositional and a planning encoding of the network components; we discussed the problems related to the planning encoding and justified our choice of focusing on the SAT encoding. We formalized both the network properties and the reconfiguration as decision problems and proved some interesting complexity results. Finally, we studied possible ways of solving the reconfiguration problem and gave an overview of the main challenges of implementing a prototype.

8.4. Acknowledgement

This work was carried on during my stay at NICTA Canberra Research Laboratory under the supervision of Jussi Rintanen and Alban Grastien, to which I'm thankful for the useful discussions and the patience they had with me.

Bibliography

[1] Qbfeval'10. `http://www.qbflib.org/index_eval.php`.

[2] Rfc 791. `http://www.faqs.org/rfcs/rfc791.html`.

[3] P. Abdulla et al. Symbolic reachability analysis based on SAT-solvers. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 411–425, 2000.

[4] E. Al-Shaer et al. Conflict classification and analysis of distributed firewall policies. *Selected Areas in Communications, IEEE Journal on*, 23(10):2069–2084, 2005.

[5] E. Al-Shaer et al. Towards global verification and analysis of network access control configuration. *DePaul University, Chicago*, 2008.

[6] E. Al-Shaer et al. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pages 123–132. IEEE, 2009.

[7] C. Baier and J.P. Katoen. *Principles of model checking*. MIT Press, 2008.

[8] M. Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In *Proc. of 20th International Conference on Automated Deduction (CADE05)*, 2005.

[9] A. Biere. Resolve and expand. In Holger Hoos and David Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 898–898. Springer Berlin / Heidelberg, 2005.

[10] A. Biere et al. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[11] R.G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[12] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, pages 333–336. Springer, 2004.

[13] A. Goultiaeva and F. Bacchus. Exploiting circuit representations in qbf solving. *LNCS-6175, Theory and Applications of Satisfiability Testing - SAT 2010*, 2010.

[14] H. Hamed et al. Modeling and verification of ipsec and vpn security policies. In *In IEEE ICNP'05*, pages 259–278, 2005.

[15] F Hüffner et al. Techniques for practical fixed-parameter algorithms. *The Computer Journal*, 51(1):7–25, 2008.

[16] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363, 1992.

[17] A. Kuehlmann et al. Circuit-based Boolean reasoning. In *Design Automation Conference, 2001. Proceedings*, pages 232–237. IEEE, 2001.

[18] Kurose and Ross. *Computer Networking - A Top-down Approach Featuring the Internet, 3rd Ed.* Addison-Wesley, 2006.

[19] P. Manolios et al. BAT: The bit-level analysis tool. In *Computer Aided Verification*, pages 303–306. Springer, 2007.

[20] S. Narain et al. Declarative infrastructure configuration synthesis and debugging. *J. Netw. Syst. Manage.*, 16:235–258, September 2008.

[21] C.H. Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[22] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177, 2010.

[23] J. Rintanen. Ai planning - lecture notes. `http://www.informatik.uni-freiburg.de/~ki/teaching/ss05/aip/s02.pdf`.

[24] J. Rintanen. Madagascar planner. `http://users.cecs.anu.edu.au/~jussi/satplan.html`.

[25] Telecordia. Ipassure webpage. `http://www.telcordia.com/products/ip-assure/index.html`, February 2011.

[26] C. Thiffault et al. Solving non-clausal formulas with dpll search. *Principles and Practice of Constraint Programming*, 2004.

[27] E. Torlak and D. Jackson. Kodkod: A relational model finder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[28] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[29] V. Vidal et al. A lookahead strategy for heuristic search planning. In *International Conference on Automated Planning and Scheduling*, pages 150–159, 2004.

A. Appendices

A.1. Directionality

In real world, firewall rules involving the direction of the information are quite common. This means that we are interested in specifying that a rule should be applied only if (e.g.) the packet is coming from subnet C and going to subnet I. This cannot be nicely modelled without considering lower level protocols. The problem is that we don't know where the packet came from when we analyse it.

To overcome this problem we could add a few more components to our expanded model. In particular, for our firewall having rules involving directionality, we have to "add" a firewall between each pair of subnets. This means that for each host we add $n*(n-1) \approx n^2$ components to our network, with n number of addresses of the host.

Note that we don't have to modify the structure of the network if we take into account a lower level protocol (i.e. MAC), since this information is encoded in the MAC address.

A.2. Disjoint Rules

We considered almost only ordered rules, but in many cases the strict order is not necessary and it can be reduced to a lattice.

A.2.1. Routers

Routing rules have a natural order based on the length of the netmask: rules with shorter netmask are applied first. Ordering these rules makes sense only if we consider the same destination prefix. In general we can express all the possible routing rules in a binary tree of depth 32, where at level d we consider the d^{th} bit of the Destination to pick the successor.

A.2.2. Firewalls

Two rules can be in one of these relations ([4]):

Completely disjoint: If every field in R_x is not a subset, superset or equal to the corresponding field in R_y

Exactly matching: If every field in R_x is equal to the corresponding field in R_y

Inclusively matching: If every field in R_x is a subset or equal to the corresponding field in R_y, and at least one is a proper subset.

Partially disjoint (matching): If there is at least one field in R_x that is a subset, superset or equal to the corresponding field in R_y, and there is at least one field in R_x that is not a subset, superset or equal to the corresponding field in R_y.

Correlated: If some fields in R_x are subsets or equal to the corresponding fields in R_y and the rest of the fields in R_x are supersets of the corresponding fields in R_y.

We are interested in detecting completely and partially disjoint rules, since they don't require ordering[1].

We need to:

- Show what sub-, superset and equality means
- Show how to check for disjointness.

In our Firewall rules we only take into account equality of fields. We don't have ranges over ports, but we do have ranges over IP addresses. The ranges over IP address are defined as the first nm bits of the address, where nm is the netmask value. We simply define for IP Addresses:

Subset: "Address A is a subset of address B if $A_{nm} \geq B_{nm}$ and A and B coincide in the first A_{nm} bits." (Obviously B is a superset of A)

Equality: "Address A is equal to address B if $A_{nm} = B_{nm}$ and A and B coincide in the first A_{nm} bits."

Disequality (disjointness): "Address A is disjoint from address B if and A and B differ for at least one bit in the first $min(A_{nm}, B_{nm})$ bits."

For ports we recall that all the rules on ranges can be rewritten as rules on single values. Therefore we can check equality and disequality in a simple way. The only kind of set relation we have for ports is the "any" superset.

Lets take two simplified rules conditions over one Address and one Port:

R1: $Address_{R1}/NM_{R1}$, "Any"

R2: $Address_{R2}/NM_{R2}$, $Port_{R2}$

The conversion of the above conditions in propositional logic is:

$\phi_{R1} = \bigwedge_{i=0}^{NM_{R1}} \sigma(Address_{R1}, i)$

$\phi_{R2} = \bigwedge_{i=0}^{NM_{R2}} \sigma(Address_{R2}, i) \wedge \bigwedge_{i=0}^{16} \sigma(Port_{R2}, i)$

with $\sigma(A, i) = A_i$ if the i-th bit of A is 1, and $\sigma(A, i) = \neg A_i$ otherwise.

Two rules are disjointed iff $(\phi_1 \wedge \phi_2) = \bot$. This can be read as the fact that this two rules don't have models (packets) in common. This kind of unsatisfiability is given by an atomic contradiction $(\neg a \wedge a)$: this comes from the definition of disjointness of an address, and from the binary encoding of the port. Thus we can check disjointness of the condition of two rules in linear time, since $\phi_1 \wedge \phi_2$ are constitute simply a big disjunction.

For the other relations we define:

Exactly matching: $\phi_1 \leftrightarrow \phi_2 = \top$

Inclusive matching: $\phi_1 \rightarrow \phi_2 = \top$

Correlated: $\exists I_1, I_2 . I_1 \models (\phi_1 \wedge \phi_2)$ and $I_2 \models (\phi_1 \leftrightarrow \neg\phi_2)$

The same ideas applies for NAT rules.

[1] If two rules have the same action, then the order is also not relevant

A.3. Possible configurations per component

The naive way of enumerating all possible configuration lines, gives us a huge number of possible states:

Router : IP/Netmask, GW = 70 bits

- Destination IP (32)
- Netmask (5) (Netmask starts from 1, the combination IP=0.0.0.0/1 means everybody)
- Next-hop (We assume that a rule to route to the interface is defined for each Address that belongs to the router)

NAT : SIP,DIP,SPort,DPort → SIP,DIP,SPort,DPort (106+96+7) = 211 bits

- Condition: SourceIP/Netmask, DestIP/Netmask, SrcPort, DestPort (32*2+6*2+16*2=108)
- Transformation: SourceIP, DestIP, SrcPort, DestPort (96)
- Position: log(108)=7

FW : SIP,DIP,SPort,DPort Accept, Reject (108+1+7) = 116 bits

- Condition: SourceIP/Netmask, DestIP/Netmask, SrcPort, DestPort (108)
- Action: Accept, reject (1)
- Position: log(108)=7

For a total of 2^{397} possible configurations for an host.

If we consider network components we can use only the maximum (encoding different informations in the same bits): 211 bits.

A.4. Finding meaningful addresses

The purpose of the reconfiguration is to set the configuration of the components in order to satisfy the policy. This means that we are not interested in encoding all the possible configurations for a network component, but only these that are meaningful to satisfy the network policy.

This means we can enumerate and number all the Addresses (IP/Netmask combination) and all the ports appearing in the policy. We use $|A|$ and $|p|$ to indicate the total number addresses and ports respectively.

With this simplification there are at most $(|A|*|p|)^4$ configuration lines per component, without taking into account the order of the rules that (for both firewalls and NAT) is important.

This number of addresses and ports is the lower bound but it might be too low to allow to configure correctly the devices. We thus need to take into account also values that appear directly in the configuration of the network: e.g. if two routers are connected to each other through a subnet S, there's no guarantee that S will appear in the policy.

This approach gives us a polynomial number of possible configurations in the number of Addresses/Ports specified in the original configuration and policy. We can consider this as an upper-bound, on the number of configuration lines and try to optimize this value on a per-component base.

Firewall reconfiguration

To better clarify the idea, we consider firewall configuration. We start by considering all the firewalls as having a deny-all policy. We are interested in allowing some traffic, maintaining the unreachability of the specified services.

We first analyse the policy and collect the fixed configurations. In particular we define 4 sets of addresses:

- SrcIP
- DstIP
- SrcPort
- DstPort

In each set we put the addresses that appear in the policy in the respective position, i.e. $A \in S$ iff $\exists p.S = A \models \tau_p$. (e.g. $80 \in DstPort$ since $DstPort = 80 \models \tau_3$ from our sample policy).

After this we enrich each set by the effects of the possible NAT translations: if there's a NAT rule whose condition contains (is a superset) of our sets, we add to our sets the effects. In order for this procedure to work we need the "transitive closure" of the NAT rules, i.e. if $A \to B$ and $C \to D$ are NAT rules and $B \subseteq C$ then $A \to D$ is a NAT rule.

This way we should obtain a set of addresses/ports that are relevant to the policy, in the context of our particular network.