NetSAT: Automated reasoning methods for verification and configuration of computer networks

Marco Gario

EMCL / NICTA

September 24, 2010

Content

Outline

Background info

- European Master in Computational Logic (FUB / TUD)
- Summer Project: 10 weeks ($\approx$2 months)
- Supervisors: Jussi Rintanen, Alban Grastien

Outline

The Problem

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).

Managing configurations in complex environments is not trivial.

Related Works

- configAssure [1] (2008):
  - Alloy modelling language → KodKod: [2] a constraint solver for relational logic
  - Complexity in the specification of the requirements as Datalog
  - KodKod solves the problem by reducing to SAT
  - Commercial product *IPAssure* by Telecordia
- ConfigChecker [3] (2009):
  - More similar to this work
  - Extension of CTL to specify requirements
  - BDD based
  - Many modelling problem (as directionality) are not explicit in the reports

Goals

- Study an alternative solution based on SAT
- Provide a working implementation
- Learn, learn, learn

Detail Level

We need to decide how much into detail we want to go. We consider only the TCP and IP level of the TCP/IP suite. Therefore we consider the following components:

- Host
- Router
- Firewall
- NAT

Outline

Network Elements

Host:

- Everything having an IP address is an Host
- At the IP level this means that everything is an Host
- can provide Services (Server) or can access them (Client)

Router:

- is an Host with at least 2 IP Addresses
- it can forward packets from one address to another based on the *RoutingTable*
- A *RoutingTable* is an ordered list of *RoutingRules*

Network Elements

Host:

- Everything having an IP address is an Host
- At the IP level this means that everything is an Host
- can provide Services (Server) or can access them (Client)

Router:

- is an Host with at least 2 IP Addresses
- it can forward packets from one address to another based on the *RoutingTable*
- A *RoutingTable* is an ordered list of *RoutingRules*

Network Elements (Cont.)

Firewall:

- is an Host that can accept or drop packets based on a *FirewallTable*
- it is usually integrated into a Router
- A *FirewallTable* is an ordered list of *FirewallRules*

NAT:

- is an Host that can modify packets based on a *NATTable*
- A *NATTable* is an ordered list of *NATRules*

Network Elements (Cont.)

Firewall:

- is an Host that can accept or drop packets based on a *FirewallTable*
- it is usually integrated into a Router
- A *FirewallTable* is an ordered list of *FirewallRules*

NAT:

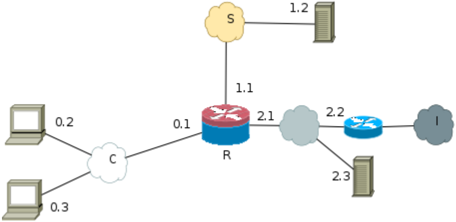
- is an Host that can modify packets based on a *NATTable*
- A *NATTable* is an ordered list of *NATRules*

Rules

	Condition	Action
Routing	Destination IP	Next Hop
Firewall	Any TCP/IP field	Accept, Deny
NAT	Any TCP/IP field	Modify any TCP/IP field

Rules are *deterministic* and are *independent* one from the other.
This structure (Condition,Action) can be used to describe the behaviour of many components in networking (eg. IPSec).

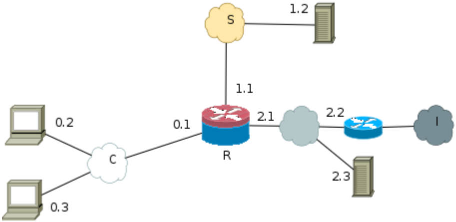
Example Network



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

R Routing table

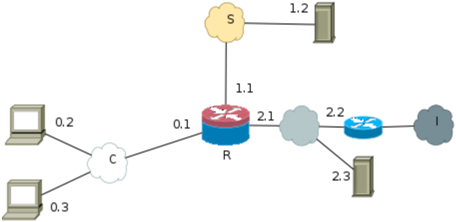
Example Network



R Firewall table

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}	Deny
SrcIP = 0.0/8 and DstIP = 2.0/8	Deny
SrcIP = 0.0/8	Accept
DstIP = 1.2/16 and DstPort = 80	Accept
SrcIP = 0.0/8 and DstIP = 1.0/8	Deny
*	Reject

Example Network



R NAT table

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

Outline

Network property

We define the following decision problem:

Basic Reachability Problem

Given:

- a network configuration $\mathcal{C}$
- an initial position Pos_0,
- a formula characterising a non-empty set of initial packets τ,
- a formula characterising the path *VALID*
- a final position Pos_n,
- and an integer n

Is it possible in the network $\mathcal{C}$ for all the packets p (s.t. $p \models \tau$) starting from Pos_0 to reach Pos_n in n steps (or less) satisfying the condition *VALID*?

Policy

We define also the Unreachability:

In the network $\mathcal{C}$ **no one** of the packets p (s.t. $p \models \tau$) starting from Pos_0 will reach Pos_n in n steps (or less) satisfying the condition *VALID*

A *Policy* is a collection of Network Properties (Reachability and Unreachability)

A *Policy* holds *iff* all the properties hold

Example

An example of policy from the previous network:

1. Nobody (except (Subnet C on port 22) and the router itself) can access the router,
2. Everybody can access port 80 on S
3. Connections to the router on port 80 should be forwarded to 1.2
4. Nobody (except from S itself) should be able to access S (except that on port 80)

Stating exceptions in a nice way is an open issue!

Example

An example of policy from the previous network:

1. Nobody (except (Subnet C on port 22) and the router itself) can access the router,
2. Everybody can access port 80 on S
3. Connections to the router on port 80 should be forwarded to 1.2
4. Nobody (except from S itself) should be able to access S (except that on port 80)

Stating exceptions in a nice way is an open issue!

Reconfiguration

We want to find a network $\mathcal{C}$ that satisfies the policy:

- We consider only configurations over the same network, that preserve the topology and the addresses;
- But we still have an huge search space (eg. $\approx 2^{200}$ possible configurations for *each* network element)
- We present a solution for a limited set of "available" configurations. How do we obtain them?

Reconfiguration

We want to find a network $\mathcal{C}$ that satisfies the policy:

- We consider only configurations over the same network, that preserve the topology and the addresses;
- But we still have an huge search space (eg. $\approx 2^{200}$ possible configurations for *each* network element)
- We present a solution for a limited set of "available" configurations. How do we obtain them?

Outline

What is not covered in this work...

- Higher / lower TCP/IP layers
- Temporal Logic
- More generic reconfiguration problem (eg, topological changes)

Outline

- Model Checking as Planning
- SAT (BMC): Gives us a shortcut on the problem, and better encoding of some properties

PDDL

- + Intuitive idea of plan as path
- + Easy “visualisation” of the path but
- - hard to extract counter-examples
- - Not all solvers accept : *requirements* like conditional-effects or disjunctive-precondition
- - Need complete solvers for Unreachability

SAT

- \+ High degree of flexibility
- \+ Many excellent solvers available
- \+ Testing the entire policy with a single SAT problem
- \+ There **is** an upperbound to the length of the solution!

QBF

We encode the reconfiguration as a 2QBF problem: $\exists\forall$.

- Not many solvers available
- Tests ran on some solvers [1] didn't terminate even when using only 1 configuration[2]
- No solver offers an easy way to extract counter examples from a $\forall\exists$ problem

Need more work on this part of the project. Using non-clausal QBF [4] solvers might help.

[1] sKizzo and quantor

[2] On the example network sKizzo crashed. Quantor didn't return any result after 30 minutes. With Minisat the same problem is solved in 2 seconds.

Outline

Overview

We consider the information contained in the packet:

- Src / Dest IP (32bit)
- Src / Dest Port (16bit)

Plus the Position of the packet in the network.

Overview (Cont.)

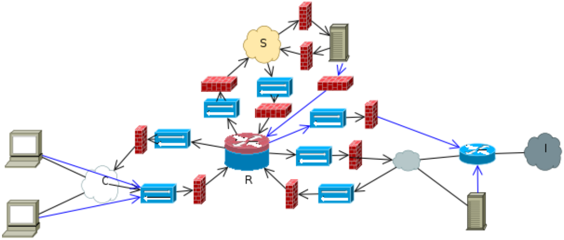
We describe the behaviour of a component with a set of planning actions/operators.

Example: Firewall

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

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

Expanded model



- We build an expanded model, in which we build a network component to represent the Firewall and the NAT of each Host.
- This components are connected in a directed graph: this way we can distinguish between incoming and outgoing paths.
- *Subnets* are added. Fictional components that behave like switches that avoid non-determinism.

From Planning to SAT

- Build the regression from the goal:
  - $A' \leftrightarrow (A \bigwedge_{(c,e) \in O s.t. \neg A \in e} \neg c) \bigvee_{(c,e) \in O s.t. A \in e} c$
- How many times should we apply the regression?
  Upperbound:
  - In IP networks we cannot have more than TTL hops $\rightarrow$ 256 hosts (ie, ≈ 1300)
  - But since the *same* packet will be processed only once by each host, we can do better than this:

    $$MP = \mathcal{O}(|Routers| * |NATRules|)$$

    between two hosts we will cross at most $|Routers|$. We can visit the same router twice only if the packet was modified, and there are $|NATRules|$ possible modifications.

From Planning to SAT

- Build the regression from the goal:
  - $A' \leftrightarrow (A \bigwedge_{(c,e)\in O s.t. \neg A \in e} \neg c) \bigvee_{(c,e)\in O s.t. A\in e} c$
- How many times should we apply the regression? Upperbound:
  - In IP networks we cannot have more than TTL hops $\rightarrow$ 256 hosts (ie, ≈ 1300)
  - But since the *same* packet will be processed only once by each host, we can do better than this:

    $$MP = \mathcal{O}(|Routers| * |NATRules|)$$

    between two hosts we will cross at most $|Routers|$. We can visit the same router twice only if the packet was modified, and there are $|NATRules|$ possible modifications.

From Planning to SAT (Cont.)

- Upperbound:
  - On our example $MP = 39$ [3]
  - Thus we use $max(1300, MP)$
- Note that the regression and the bound are independent from the properties that we want to check!

[3] The exact formula is $MP = 5(|Router| * |NATRules| + 1) + |NATRules| + 1$

From Planning to SAT (Cont.)

- Upperbound:
  - On our example $MP = 39$ [3]
  - Thus we use $max(1300, MP)$
- Note that the regression and the bound are independent from the properties that we want to check!

[3] The exact formula is $MP = 5(|Router| * |NATRules| + 1) + |NATRules| + 1$

Properties

From the Basic Reachability Problem we define 5 properties that we are interested in solving:

	τ	*VALID*
P1	1 Value per field	$\top$
P2	1 Value per field	Arbitrary over $Visited_i$[4] and PKT_n
P3	$\top$	$\top$
P4	Arbitrary over PKT_0	$\top$
P5	Arbitrary over PKT_0	Arbitrary over $Visited_i$ and PKT_n

- P1 Specific (Un)Reachability
- P2 Element traversal
- P3 Full (Un)Reachability
- P4 Quantified (Un)Reachability
- P5 Quantified element traversal ($= BRP$)

[4] This proposition was introduced to denote the fact that "at some point in time" the component i was visited

Properties (Cont.)

Verifying the properties means solving the following TAUT problems:

P1, P2	$\phi_{P2} = I_0 \rightarrow (G_n \wedge VALID)^n$
P3, P4, P5	$\phi_{BRP} = \forall PKT_0.(\tau(PKT_0) \wedge Pos_0) \rightarrow (Pos_n \wedge VALID(Pos_i, PKT_n))^n$

All this properties have linear complexity!
If we extend the quantification to all starting positions we obtain

$$\phi^*_{BRP} = \forall Pos_0 \forall PKT_0.\sigma(Pos_0, PKT_0) \wedge \phi_{BRP}(PKT_0)$$

that is not linear anymore but $\in coNP$.[5]

We can use a SAT solver to verify UNSAT of $\neg\phi^*_{BRP}$

[5] σ relates the starting positions with IP addresses

Precompute the regression

In a policy with k properties we need to solve k UNSAT problems. We can use more efficiently the SAT solver by building one problem for the whole policy:

- We can compute, by means of the regression, a formula describing the relation between the initial and final "states" (RT).
- Since this depends only on the configuration of the network, we can use it for testing multiple properties!
- We build a new UNSAT problem:

$$\exists Pos_0, PKT_0.RT \wedge (P_i \vee \ldots \vee P_k)$$

with $P_i = \tau \wedge \sigma \wedge Pos_0 \wedge \neg(Pos_n \wedge VALID)$

Reconfiguration

Can we modify the network components configuration in order to satisfy the Policy?

$$\exists c_0, .., c_m . \forall Pos_0, PKT_0 . RT \rightarrow \neg(P_1(c_0, ..., c_m) \vee ... \vee P_k(c_0, ..., c_m))$$

where $c_0, .., c_m$ are the configuration parameters for all the components.
Recall that P_i describes the violation of the Property.

Reconfiguration (Cont.)

Problems:

- This formulation of the problem gives a huge search space!
- Lets define a set $\mathcal{C}$ of possible configurations. We assume $\mathcal{C}$ is provided.
- Couldn't solve the problem with a standard QBF solver! Implementative details makes it a 3QBF $\exists\forall\exists$!

In this simplified scenario we can use an incremental SAT solver and perform a linear search. But in general we think this problem to be Σ_2^P-hard.

Reconfiguration (Cont.)

Problems:

- This formulation of the problem gives a huge search space!
- Lets define a set $\mathcal{C}$ of possible configurations. We assume $\mathcal{C}$ is provided.
- Couldn't solve the problem with a standard QBF solver! Implementative details makes it a 3QBF $\exists\forall\exists$!

In this simplified scenario we can use an incremental SAT solver and perform a linear search. But in general we think this problem to be Σ_2^P-hard.

Reconfiguration (Cont.)

Problems:

- This formulation of the problem gives a huge search space!
- Lets define a set $\mathcal{C}$ of possible configurations. We assume $\mathcal{C}$ is provided.
- Couldn't solve the problem with a standard QBF solver!
  Implementative details makes it a 3QBF $\exists\forall\exists$!

In this simplified scenario we can use an incremental SAT solver and perform a linear search. But in general we think this problem to be Σ_2^P-hard.

Reconfiguration (Cont.)

Problems:

- This formulation of the problem gives a huge search space!
- Lets define a set $\mathcal{C}$ of possible configurations. We assume $\mathcal{C}$ is provided.
- Couldn't solve the problem with a standard QBF solver! Implementative details makes it a 3QBF $\exists\forall\exists$!

In this simplified scenario we can use an incremental SAT solver and perform a linear search. But in general we think this problem to be Σ_2^P-hard.

Outline

Implementation info

The tool to solve this problem was developed in Java, using SAT4J / Minisat as SAT solvers.

In the process I developed a library to:

- Generate the PDDL domain and problem.
- Manipulate big formulae as circuit,
- Convert from/to DIMACS and manipulate directly the DIMACS CNF,
- Build the regression and simplify it

There's lots of space for improvement!

Implementation info

The tool to solve this problem was developed in Java, using SAT4J / Minisat as SAT solvers.

In the process I developed a library to:

- Generate the PDDL domain and problem.
- Manipulate big formulae as circuit,
- Convert from/to DIMACS and manipulate directly the DIMACS CNF,
- Build the regression and simplify it

There's lots of space for improvement!

Circuits for formulae

We deal with big formulae:

- Each equality $(A' \leftrightarrow \phi)$ potentially depends on all the other components of the network, meaning
- Each equality might contain all the propositional variables: regression is exponential in the number of the variables!

Compact representation of the formulae: Circuit.

+ Reuse common subformulae $\rightarrow$ usefull when making substitutions
+ Trivial to perform the Tseitin conversion
- Not many studies on what type of Circuit behaves “better” (RBC [6], AIG, NICE [5], etc.)

+/- Keeping the formula as a Circuit allows some enhanced reasoning that is not possible in CNF (eg. Don't care [7])

Circuits for formulae

We deal with big formulae:

- Each equality ($A' \leftrightarrow \phi$) potentially depends on all the other components of the network, meaning
- Each equality might contain all the propositional variables: regression is exponential in the number of the variables!

Compact representation of the formulae: Circuit.

+ Reuse common subformulae $\rightarrow$ usefull when making substitutions
+ Trivial to perform the Tseitin conversion
- Not many studies on what type of Circuit behaves "better" (RBC [6], AIG, NICE [5], etc.)

+/- Keeping the formula as a Circuit allows some enhanced reasoning that is not possible in CNF (eg. Don't care [7])

Tseitin conversion: A practical problem

- We need to convert the problem ϕ into CNF to give it to the SAT solver
- This is usually performed by Tseitin conversion $\rightarrow$ Equisat formula ϕ^*_{CNF} in CNF in Linear time by using auxiliary variables!
- There's a catch! We need to test Tautology of ϕ but we cannot do it directly on ϕ^*_{CNF}!
- This becomes a problem when trying to solve the reconfiguration problem: $\exists Conf \forall Init \exists aux \phi^*_{CNF}$
- From 2QBF to 3QBF!

The aux variables are the reason why the QBF solver cannot solve this problem!

Tseitin conversion: A practical problem

- We need to convert the problem ϕ into CNF to give it to the SAT solver
- This is usually performed by Tseitin conversion $\rightarrow$ Equisat formula ϕ^*_{CNF} in CNF in Linear time by using auxiliary variables!
- There's a catch! We need to test Tautology of ϕ but we cannot do it directly on ϕ^*_{CNF}!
- This becomes a problem when trying to solve the reconfiguration problem: $\exists Conf \forall Init \exists aux \phi^*_{CNF}$
- From 2QBF to 3QBF!

The aux variables are the reason why the QBF solver cannot solve this problem!

Tseitin conversion: A practical problem

- We need to convert the problem ϕ into CNF to give it to the SAT solver
- This is usually performed by Tseitin conversion $\rightarrow$ Equisat formula ϕ^*_{CNF} in CNF in Linear time by using auxiliary variables!
- There's a catch! We need to test Tautology of ϕ but we cannot do it directly on ϕ^*_{CNF}!
- This becomes a problem when trying to solve the reconfiguration problem: $\exists Conf \forall Init \exists aux \phi^*_{CNF}$
- From 2QBF to 3QBF!

The aux variables are the reason why the QBF solver cannot solve this problem!

Running time examples

In our example network we add 2 possible configurations that don't satisfy the policy. Here's the output of the tool on the Example network. Property 4 is:
Reachability from 192.168.0.0/24 (ClientNetwork) to 192.168.1.2:80

Outline

Really interesting ones

- How to generate the configuration set $\mathcal{C}$
- Proper proofs for the complexity results
- Disjointness of rules
- Scaling Tests
- Higher levels with more complex interactions (eg. TCP sessions)

Out of scope

- Lower levels
- Comparison with “existing” tools
- Static vs Dynamic configuration

Outline

Critics

Some things that I notice during my stay:

- I need to be able to anticipate theoretical problem earlier since
- lack of out-of-the-box tools diverts lots of time on implementation of “not-so-relevant” things.
- Quadratic is fine in theory, but with such huge formulae it becomes not practical.
- I didn’t had the goal too clear in my mind when I started, so I look at many related problems and now I have many open issues

Definitely learned a lot of things and have a working tool that solves the problem.

Comments

- Questions?
- Critics?
- Suggestions?

I'm a Master student, I need them!

References

- Narain, S. and Levin, G. and Malik, S. and Kaul, V., Declarative infrastructure configuration synthesis and debugging, Journal of Network and Systems Management, 2008
- http://alloy.mit.edu/kodkod/
- Ehab Al-Shaer, Will Marrero, Adel El-Atawy and Khalid Elbadawi, Network Security Configuration in A Box: End-to-End Security
- Goultiaeva, Alexandra and Bacchus, Fahiem, Exploiting Circuit Representations in QBF Solving, LNCS-6175 Theory and Applications of Satisfiability Testing - SAT 2010, 2010.
- P Manolios, S Srinivasan, D Vroon, BAT: The bit-level analysis tool, Computer Aided Verification, 2007 - Springer