2008-01-24

Examples of Network Flow Verification Using TRAFFIC(X)

Examples of Network Flow Verification Using TRAFFIC($\mathbf{X}$)*

LIKAI LIU liulk@cs.bu.edu
ASSAF KFOURY kfoury@cs.bu.edu

Computer Science Department
Boston University

January 24, 2008

Technical report: BUCS-TR-2007-017

Abstract

In our previous work, we developed TRAFFIC($\mathbf{X}$), a specification language for modeling bi-directional network flows featuring a type system with constrained polymorphism. In this paper, we present two ways to customize the constraint system: (1) when using linear inequality constraints for the constraint system, TRAFFIC($\mathbf{X}$) can describe flows with numeric properties such as MTU (maximum transmission unit), RTT (round trip time), traversal order, and bandwidth allocation over parallel paths; (2) when using Boolean predicate constraints for the constraint system, TRAFFIC($\mathbf{X}$) can describe routing policies of an IP network. These examples illustrate how to use the customized type system.

1 Introduction

Experimenting with complex network systems can be expensive, both in the cost of hardware and in the labor to configure physical network nodes. As physical units are built to have increasingly better durability and performance, we can now build networks that support higher-level applications, such as virtual private network (VPN), video streaming network in 3G, and network for distributed computing. These applications may even span several link layer technologies where encapsulation is commonly used for interoperability (e.g., IP over over GRE/IP over Ethernet over ATM over SONET). Diagnosis now becomes problematic because errors may be introduced due to interaction of several units or layers when they are combined, rather than due to the fault of individual units.

Network Simulators [1] let a network designer run a network model on a computer and observe its behavior, thus reducing cost of experimentation. However, testing alone does not confer correctness. The notion of correctness also needs to be defined, usually by listing functional properties of the system and making sure they hold for the model.

Formal verification is a practice that models a system in the syntax of an abstract language and uses mathematical expressions to state functional properties. Although the cost benefit is not immediately obvious, this aids design by formulating a concrete specification of a system. The end result is a machine checked specification that avoids human error factor.

Related Work Many formal verification frameworks have been proposed. Petri net [14, 13] and π-calculus [9] are traditionally used to model concurrency properties of a system. These are interesting to us because networks can be seen as concurrent systems, and some network properties can be encoded in terms of known analysis, taking advantage of extensive research in these areas. Theorem proving (Isabelle[11], Twelf [15], PVS [12], and Coq [7]) has been used to prove the correctness of an ATM network switching fabric [4], albeit through significant manual effort. Binary Decision Diagram [2] has been used to detect policy conflicts of IPsec rules [6] with some success, and it inspires the example in Section 3.

*This work is partially supported by NSF Award No. CCR-0205294

$$
\begin{array}{rcll}
x, y, z & \in & \mathsf{FlowVar} & \text{flow variable} \\
A, B & \in & \mathsf{LocalFlow} & \text{local flow} \\
\mathcal{A}, \mathcal{B} & \in & \mathsf{GlobalFlow} ::= x \mid A & \\
& & \mid \mathcal{A} ; \mathcal{B} & \text{sequential flow} \\
& & \mid \mathcal{A} \parallel \mathcal{B} & \text{parallel flow} \\
& & \mid \mathbf{let}\ x = \mathcal{A}\ \mathbf{in}\ \mathcal{B} & \text{let-binding}
\end{array}
$$

Figure 1.1: Syntax of flow specification.

1.1 The TRAFFIC(X) Framework

We developed TRAFFIC(**X**), which is both a specification language and a type system that serves as the basis for verification [8], to specialize in high-level networked applications. The global-flow specification language provides a way to model end-to-end network *flow*, a bi-directional stream of data, which is the basic functional abstraction in our model. Two flows can be put together sequentially or in parallel, which results in a larger flow. The smallest unit of a flow is a *local flow*, which represents off-the-shelf components with known properties.

The language also has a "let-binding" syntax that provides an abstraction, so a programmer can define a flow as some variable once and use the flow many times by referring to the variable. An important property that constitutes the soundness of our type system stems from the fact that using let-binding does not change the correctness of a flow. A flow with let-bindings can be normalized to one without, and the resulting flow is type-safe if and only if the original flow is type-safe.

Assigning Types to Flow A flow has four ports—inputs and outputs of forward and backward directions—and this is reflected in the flow type, written as $\left[\begin{smallmatrix} \rho_i & \rho_o \\ \sigma_o & \sigma_i \end{smallmatrix}\right]$, where ρ is for forward type, and σ for backward type; subscript i is for input, and subscript o for output. A flow type describes interfacing properties of a flow and determines how one flow may be composed with another. To illustrate sequential and parallel composition, suppose we have flows $\mathcal{A} : \left[\begin{smallmatrix} \rho_i & \rho_o \\ \sigma_o & \sigma_i \end{smallmatrix}\right]$ and $\mathcal{A}' : \left[\begin{smallmatrix} \rho'_i & \rho'_o \\ \sigma'_o & \sigma'_i \end{smallmatrix}\right]$.

When combining two flows in parallel, no connection is done; all ports of the two flows are simply paired together respectively, and we expose four pairs of bundled ports as $(\mathcal{A} \parallel \mathcal{A}') : \left[\begin{smallmatrix} \rho_i \cdot \rho'_i & \rho_o \cdot \rho'_o \\ \sigma_o \cdot \sigma'_o & \sigma_i \cdot \sigma'_i \end{smallmatrix}\right]$.

When combining two flows sequentially, forward output of the first flow is connected to the forward input of the second, and backward output of the second flow is connected to the backward input of the first. The type system verifies, according to type of these flows, whether such connection is safe to do. Generally, we check if output and input have equal types. Sometimes we can relax the requirement by checking if output type is a *subtype* of input type, meaning it is okay to provide data that over-qualify for the requirement. Once this is done, we only need to expose the outer connections, and we can regard the sequentially composed flows as being one flow $(\mathcal{A} ; \mathcal{A}') : \left[\begin{smallmatrix} \rho_i & \rho'_o \\ \sigma_o & \sigma'_i \end{smallmatrix}\right]$.

Flow types of local flows are defined by the user. We use the notation $\mathsf{type}(A) = \left[\begin{smallmatrix} \rho_i & \rho_o \\ \sigma_o & \sigma_i \end{smallmatrix}\right]$ to denote the assignment of flow type $\left[\begin{smallmatrix} \rho_i & \rho_o \\ \sigma_o & \sigma_i \end{smallmatrix}\right]$ to local flow A.

Type Variables and Constraints A distinct feature of TRAFFIC(**X**) is to allow types to be partially specified; any unspecified part is written as a type variable. Two partially specified types can be related by using the same type variable. In the presence of type variables, checking if two types are equal becomes the question whether there exists a way to substitute type variables for concrete types that makes the two types equal—this is the notion of *equality constraint*. Similarly, checking whether there exists a way to substitute type variables so a type is a subtype of another type is the notion of *subtyping constraint*.

When flows with partially specified types are composed sequentially, equality and subtyping constraints are collected and considered altogether. A set of constraints is *satisfiable* if there exists a substitution for all the type variables in the constraints so that all equality and subtyping relations in the set hold.

Polymorphism with Constraints Type variables that appear in a flow type that do not rely on external assumption can be generalized using the $\forall$-quantifier. For this, we introduce flow type scheme written as $\forall \bar{\alpha}(C). \left[\begin{smallmatrix} \rho_i & \rho_o \\ \sigma_o & \sigma_i \end{smallmatrix}\right]$, where $\bar{\alpha} = (\alpha_1, \ldots, \alpha_n)$ is a vector of type variables that appear in $\rho_i, \rho_o, \sigma_o, \sigma_i$. We also have C, a set of constraints, to

$$
\begin{array}{rcll}
r, s, t & \in & \mathsf{TypeLit} & \\
\alpha, \beta & \in & \mathsf{TypeVar} & \\
F & \in & \mathsf{TypeCons} & \\
\rho, \sigma, \tau & \in & \mathsf{Type} & ::= \quad t \mid \alpha \mid \tau_1 \cdot \tau_2 \mid F\ \tau_1 \ldots \tau_n \\
T & \in & \mathsf{FlowType} & ::= \quad \left[\begin{smallmatrix} \rho_i & \rho_o \\ \sigma_o & \sigma_i \end{smallmatrix}\right] \\
 & & & \\
P & \in & \mathsf{Predicate} & \\
c, d & \in & \mathsf{Constraint} & ::= \quad \tau_1 <: \tau_2 \mid \tau_1 \doteq \tau_2 \mid P\ \tau_1 \ldots \tau_n \mid \mathit{true} \mid \mathit{false} \\
 & & & \quad\ \mid \quad T_1 <: T_2 \mid T_1 \doteq T_2 \mid P\ T_1 \ldots T_n \\
C, D, E & \in & \mathsf{ConstraintSet} & ::= \quad \{c_1, \ldots, c_n\} \\
 & & & \\
S & \in & \mathsf{FlowTypeScheme} & ::= \quad T \mid \forall \bar{\alpha}(C).T
\end{array}
$$

Figure 1.2: Syntax of Types

act as a guard. The flow type scheme is instantiated by substituting $\bar{\alpha}$ for either fresh type variables or concrete types, provided the constraint set C subject to the same substitution remains satisfiable.

User never manipulates type variables directly. The type system, defined in Appendix A, serves as a vehicle to collect, generalize, and instantiate constraints by walking the syntax case-by-case. Manipulation of type variables and constraints is done by a constraint solver. As we promote more prominent use of type with type variables, type system becomes an interface to the constraint solver, which does the actual verification.

1.2 Customizing $\mathsf{TRAFFIC}(\mathbf{X})$

Summarizing the explanation above, syntax of types and variable use is formally defined in Figure 1.2. So far, we have seen α, β to range over the set of type variables, or TypeVar; ρ, σ, τ for types that describe a port, or Type; and C, D for constraint set. We use T to range over the set of flow types, or FlowType, and S to range over flow type schemes. The remaining items that deserve our attention are TypeLit (type literals), TypeCons (type constructors), and Predicate (constraint predicates), and the definition and interpretation of these items vary from application to application, depending on how $\mathsf{TRAFFIC}(\mathbf{X})$ is used to perform formal verification. These customizations are the $\mathbf{X}$ in $\mathsf{TRAFFIC}(\mathbf{X})$.

Type literals are the most basic description of a type. For example, when describing bandwidth usage of a port, it would be a number. Type constructors are abstract functions that take types as input and return a type, i.e., $\mathsf{Type}^n \to \mathsf{Type}$ for an n-ary type constructor. The bundling operator for parallel flows $(\cdot)$ is a special case of type constructor. Predicates are functions with the sort $\mathsf{Type}^n \to \mathsf{Constraint}$ for an n-ary predicate. The special cases are subtyping $(<:)$ and equality $(\doteq)$ predicates. There are also the nullary predicates true and false.

The only operation required from a constraint solver is the entailment relation $(\Vdash)$; that is, given two constraint sets C and D, decide if $C \Vdash D$. Conceptually, C entails D means that C is a stricter set of constraints than D, so that C is satisfiable implies D is satisfiable.

Constraint solver must be able to resolve constraints using every predicates defined in Predicate, so introduction of new predicates requires modifying the constraint solver to handle them. The constraints may also contain new type constructors, and the constraint solver must be extended accordingly. Since implementation and changes to constraint solver could be rather involved, we expect that type constructors, predicates, and constraint solver to be provided as a whole package $\mathbf{X}$ for $\mathsf{TRAFFIC}(\mathbf{X})$ to the user.

In this paper, we examine two ways to customize types and constraint system and provide some examples to illustrate what they are useful for.

2 Linear Inequality Constraints

Types as numeric quantities can be used to represent bandwidth, bandwidth delay product, maximum transmission unit (MTU), number of concurrent connections, and TCP window size. When considering a numeric type with type variables, it is natural to extend types to include algebraic operators so we can write a type as a function of some

variables. An inequality constraint results from comparing two numeric quantities with unknowns. A linear inequality constraint consists of constants and degree one variables with constant coefficients.

2.1 Defining Constraint System

Let TypeLit range over real numbers, and we introduce three binary type constructors $+$, $-$ and $\times$. The resulting Type allows us to construct polynomials whose variables range over TypeVar. The meaning of subtyping ($<:$) is treated as if it were less-than-or-equal-to ($\leq_{\mathbb{R}}$), and equality ($\doteq$) the same as algebraic equality ($=_{\mathbb{R}}$). Even though the way we define Type can potentially result in non-linear polynomials, our constraint solver will only accept linear constraints.

We are only interested to see if a set of constraints have a feasible solution, which can be determined using the initialization routine of the Simplex algorithm [3, pp. 812]. Special caution must be observed that variables of a feasible solution must have non-negative values.

A *feasible region* of a set of constraints is the space enclosed by all the inequalities in that set. We write $C \Vdash D$ to denote that the feasible region defined by constraints in C lies entirely inside the feasible region defined by constraints in D. If C is infeasible, then $C \Vdash \mathit{false}$. It should be clear that false represents the empty region, and true represents the entire unbounded quadrant.

To decide if $C \Vdash D$, we consider the base case $C \Vdash d$ supposing there is only one constraint d in D. Let I be a set of points that constraints in C intersect each other, and J be a set of points that constraints in C intersect d. If there is a point in $J - I$ that lies inside the feasible region of C, then we know d "cuts off" a part of C, so $C \not\Vdash d$. We iterate this process for all constraints in D, that is, $C \Vdash D$ if $C \Vdash d$ for all $d \in D$.

2.2 Path Maximum Transmission Unit

As an introduction for how constraints are collected and processed, let us consider the scenario for maximum transmission unit.

Data are sent over a network as packets, which are parsed, processed, and retransmitted as they traverse through routers and switches. For reasons such as collision control and buffering, these packets have a size limit, which is called maximum transmission unit (MTU). It is common to have heterogeneous network where individual segments are built from different link layer technologies, hence have different packet size limits. IP (Internet Protocol) deals with this situation by allowing packets to be fragmented. However, fragmentation is undesirable because it increases the chance of packet loss. Fragments may also arrive out of order, making reassembly difficult. Path MTU discovery is a technique to avoid IP packet fragmentation [10] by finding the smallest MTU along a point-to-point path.

In this example, we show how to describe MTU constraints in $\mathsf{TRAFFIC}(\mathbf{X})$ and explain how these constraints are collected. Suppose we use the following local flows to represent various network segments with different MTUs.

Local Flow	MTU
A_1	2000
A_2	1100
A_3	1500

For each local flow A_i listed above, we have

$$\mathsf{type}(A_i) \quad = \quad \forall p, q(p, q \leq \mu_i). \begin{bmatrix} p & p \\ q & q \end{bmatrix}$$

where μ_i is the MTU for A_i. Types describe the packet size. As indicated, the incoming packet size is exactly the same as outgoing packet size.

An end-to-end flow passing through all these segments is expressed as $A_1 ; A_2 ; A_3$, so we apply (SEQ) rule twice. The type system has to first instantiate the flow type schemes and turn p and q to fresh type variable instances. We get the following flow type diagram as the result.

$$\underset{A_1}{\begin{bmatrix} p_1 & p_1 \\ q_1 & q_1 \end{bmatrix}} \quad - \quad \underset{A_2}{\begin{bmatrix} p_2 & p_2 \\ q_2 & q_2 \end{bmatrix}} \quad - \quad \underset{A_3}{\begin{bmatrix} p_3 & p_3 \\ q_3 & q_3 \end{bmatrix}}$$

This flow has the type $\left[\begin{smallmatrix} p_1 & p_3 \\ q_1 & q_3 \end{smallmatrix}\right]$, and the constraints to be solved are $\{p_1 \doteq p_2, p_2 \doteq p_3, q_3 \doteq q_2, q_2 \doteq q_1\} \cup \{p_1, q_1 \leq 2000\} \cup \{p_2, q_2 \leq 1100\} \cup \{p_3, q_3 \leq 1500\}$. Altogether, this means that all the p's and q's must be less than or equal to 1100.

2.3 Computing Round Trip Time

In this example, we demonstrate how latency can be expressed in flow types, and how this information can be used to compute round trip time using linear inequality constraints. Suppose the following local flows have latencies that are asymmetric for forward and backward directions:

Local Flow	Forward Lat. ϕ	Backward Lat. δ
A_1	10 ms.	6 ms.
A_2	12 ms.	15 ms.
A_3	8 ms.	9 ms.

For each local flow A_i listed above, we have

$$\mathsf{type}(A_i) \quad = \quad \forall x, y. \begin{bmatrix} x & x+\phi_i \\ y+\delta_i & y \end{bmatrix}$$

where ϕ_i is the forward latency, and δ_i the backward latency. Types describe the wall clock time a signal enters or leaves a flow, in milliseconds. The flow type is quantified over two variables, x, which denotes the time a signal enters the forward input, and y, which denotes the time a signal enters the backward input. Forward signal exits at time $x + \phi_i$, and backward signal at time $y + \delta_i$.

To compute round trip time of the flow $A_1 \,;\, A_2 \,;\, A_3$, we need two more pieces: $Ping$ initiates a probe signal, and $Pong$ returns it at the far end. To make the example more interesting, suppose $Pong$ also inflicts a latency of 50 ms. for returning the signal. Both local flows are simply schematic; we do not simulate the sending of probe signal, but rely on the constraint solver to analyze statically whether a given round-trip time can be satisfied. Flow type schemes of $Ping$ and $Pong$ are defined as follows:

$$\mathsf{type}(Ping) \quad = \quad \forall x, y(D). \begin{bmatrix} F_{\mathsf{nil}} & x \\ F_{\mathsf{nil}} & y \end{bmatrix} \qquad \mathsf{type}(Pong) \quad = \quad \forall x. \begin{bmatrix} x & F_{\mathsf{nil}} \\ x+50 & F_{\mathsf{nil}} \end{bmatrix}$$

The additional constraints for limiting round-trip time is added to the constraints of $Ping$, in set D. An example is to let $D = \{y - x \leq 100\}$, i.e., the time difference between sending a probe signal and receiving it back is less than or equal to 100 ms. One can immediately compute that such round-trip time cannot be satisfied in this setting. However, let us examine how the constraint solver comes to the same conclusion.

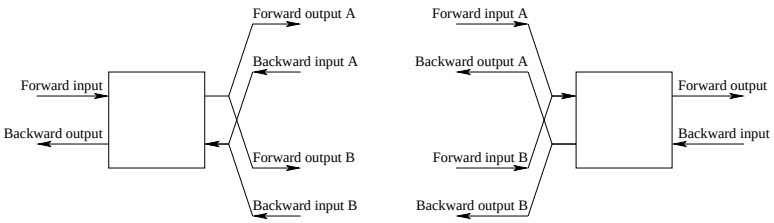
First, the type system of $\mathsf{TRAFFIC}(\mathbf{X})$ causes flow type schemes of $Ping, A_1, A_2, A_3, Pong$ to be instantiated. The x and y variables become distinct variables, so we obtain the following flow type diagram:

$$\underset{Ping}{\begin{bmatrix} F_{\mathsf{nil}} & x_0 \\ F_{\mathsf{nil}} & y_0 \end{bmatrix}} \; - \; \underset{A_1}{\begin{bmatrix} x_1 & x_1+10 \\ y_1+6 & y_1 \end{bmatrix}} \; - \; \underset{A_2}{\begin{bmatrix} x_2 & x_2+12 \\ y_2+15 & y_2 \end{bmatrix}} \; - \; \underset{A_3}{\begin{bmatrix} x_3 & x_3+8 \\ y_3+9 & y_3 \end{bmatrix}} \; - \; \underset{Pong}{\begin{bmatrix} x_4 & F_{\mathsf{nil}} \\ x_4+50 & F_{\mathsf{nil}} \end{bmatrix}}$$

This flow is a sequential flow, so (SEQ) rule is applied (several times in succession) to produce a typing of this flow. The final type of the flow is $\left[\begin{smallmatrix} F_{\mathsf{nil}} & F_{\mathsf{nil}} \\ F_{\mathsf{nil}} & F_{\mathsf{nil}} \end{smallmatrix}\right]$, which is not very interesting, but the rule requires the following constraints to be solved:

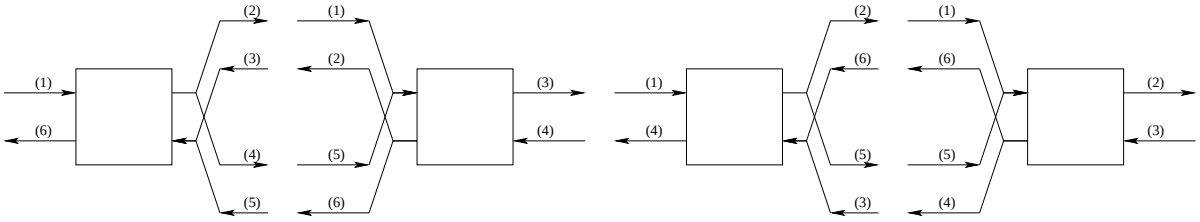
$$\begin{array}{ll} \{ & x_0 = x_1,\ x_1 + 10 = x_2,\ x_2 + 12 = x_3,\ x_3 + 8 = x_4,\ x_4 + 50 = y_3, \\ & y_3 + 9 = y_2,\ y_2 + 15 = y_1,\ y_1 + 6 = y_0,\ y_0 - x_0 \leq 100 \qquad\qquad \} \end{array}$$

At this moment, the simplex solver is not invoked yet, because many of those equality constraints can be eliminated by simple substitution. It boils down to $y_0 = x_0 + 110$, which expresses the exact relation between start and end time. We then have the constraint $x_0 + 110 - x_0 \leq 100$, and after algebraic simplification, we have $110 \leq 100$, which is false.



(a) Splitter (b) Combiner

Figure 2.1: Splitter and combiner flows.



(a) L_1 (b) R_1 (c) L_2 (d) R_2

Figure 2.2: Two possible packet traversal order assignments.

2.4 Packet Traversal Order

Taking the previous example further, we can use constraints to restrict the order of packet traversal. For every local flow A_i as in the previous section, we can express the requirement that packets exit in the forward direction before entering in the backward direction by writing the flow type as:

$$\mathsf{type}(A_i) \quad = \quad \forall x, y\ (x + \phi_i \leq y).\ \begin{bmatrix} x & x + \phi_i \\ y + \delta_i & y \end{bmatrix}$$

where ϕ_i is the forward latency, and δ_i the backward latency; $x + \phi_i$ is the forward exit time, and y the backward arrival time.

So far, flows have been composed sequentially, and we have not examined cases for parallel flow. Two parallel flows are by default independent of each other, unless they are made to interact using combiner or splitter as shown in Figure 2.1. Splitter and combiner let us divide one flow path into two and later join them.

Determining round trip time in the presence of multiple flow paths becomes an interesting problem because there are many ways that a packet can be routed over multiple paths. Figure 2.2 shows two possible order assignments $L_1 \,;\, R_1$ and $L_2 \,;\, R_2$, where the numeric labels denote packet traversal order.

Using σ for forward type and ρ for backward type, the flow type of a splitter is written as $\begin{bmatrix} \sigma_{\mathsf{in}} & (\sigma_{\mathsf{out.A}} \cdot \sigma_{\mathsf{out.B}}) \\ \tau_{\mathsf{out}} & (\tau_{\mathsf{in.A}} \cdot \tau_{\mathsf{in.B}}) \end{bmatrix}$, and a combiner as $\begin{bmatrix} (\sigma_{\mathsf{in.A}} \cdot \sigma_{\mathsf{in.B}}) & \sigma_{\mathsf{out}} \\ (\tau_{\mathsf{out.A}} \cdot \tau_{\mathsf{out.B}}) & \tau_{\mathsf{in}} \end{bmatrix}$. If we name variables x_k in the order they are seen at each socket position, we can write the flow type schemes for $L_1\ R_1\ L_2\ R_2$ as follows:

$$\mathsf{type}(L_1) \quad = \quad \forall \bar{x}(C).\ \begin{bmatrix} x_1 & (x_2 \cdot x_4) \\ x_6 & (x_3 \cdot x_5) \end{bmatrix} \qquad \mathsf{type}(R_1) \quad = \quad \forall \bar{x}(C).\ \begin{bmatrix} (x_1 \cdot x_5) & x_3 \\ (x_2 \cdot x_6) & x_4 \end{bmatrix}$$

$$\mathsf{type}(L_2) \quad = \quad \forall \bar{x}(C).\ \begin{bmatrix} x_1 & (x_2 \cdot x_5) \\ x_4 & (x_6 \cdot x_3) \end{bmatrix} \qquad \mathsf{type}(R_2) \quad = \quad \forall \bar{x}(C).\ \begin{bmatrix} (x_1 \cdot x_5) & x_2 \\ (x_6 \cdot x_4) & x_3 \end{bmatrix}$$

and we simply let $C = \{x_i \leq x_{i+1} \mid 1 \leq i \leq 5\}$.

Notice that if we mix the two order assignments, i.e. $L_1 \,;\, R_2$ and $L_2 \,;\, R_1$, then antisymmetry of $\leq$ forces packets to appear at exact the same time on multiple paths. If the two paths both incur latency, then the resulting constraint sets would not be solvable.

2.5 Parallel Flow Bandwidth

Accounting bandwidth usage, when flow is split and combined, is also an interesting use of linear inequality constraints. When flows are combined, their bandwidth uses are added together. Constraints allow us to check whether the total bandwidth does not exceed a certain quantity.

For example, suppose A_1 and A_2 are gateways connecting two networks. Schematically, A_1 is a combiner in Figure 2.1, and A_2 is a splitter. The flow types of their bandwidth usage can be written as follows:

$$\mathsf{type}(A_1) = \forall \bar{a}, \bar{b}. \begin{bmatrix} (a_1 \cdot a_2) & a_1 + a_2 \\ (b_1 \cdot b_2) & b_1 + b_2 \end{bmatrix} \quad \mathsf{type}(B) = \forall a, b(C). \begin{bmatrix} a & a \\ b & b \end{bmatrix} \quad \mathsf{type}(A_2) = \forall \bar{a}, \bar{b}. \begin{bmatrix} a_1 + a_2 & (a_1 \cdot a_2) \\ b_1 + b_2 & (b_1 \cdot b_2) \end{bmatrix}$$

There are several possibilities to write the constraint set C. These bandwidth accounting methods have different implications on the limits of network traffic pattern.

- If bandwidths of the forward and backward directions are independently throttled, for example, at 10 Mbps and 5 Mbps respectively, then $C = \{a \leq 10, b \leq 5\}$. This scenario is commonly seen for ADSL (Asymmetric Digital Subscriber Line) connections, where the asymmetry is in downstream and upstream bandwidth allowance.
- If bandwidths of the forward and backward directions altogether may not exceed, for example, 15 Mbps, then $C = \{a + b \leq 15\}$. This scenario is commonly seen for leased, dedicated line subscriptions to an ISP.

Furthermore, the delivery medium between A_1 and A_2 may specify more bandwidth constraints.

3 Boolean Predicate Constraints

A network typically allows all sorts of packets to be transmitted from any node to another. However, upon closer inspection, there are a number of situations where nonsensical packets should be dropped; such packets are typically also non-routable. For example,

- Packets originating from or destined to $127.0.0.0/8$ are non-routable. Such packets should never appear on a network.
- Packets traversing from network A to network B should have the source IP that matches the subnet of network A and destination IP that matches the subnet of network B. Vice versa for packets traversing from network B to network A.
- Every host has an interface with an IP address. All packets originating from this interface must have its source IP address the same as the interface IP address. Only packets destined for this interface address may be received by the host.

Traditionally, networks are designed with best-effort delivery with no access control; nonsensical packets may still be delivered as long as it has a destination address that can be recognized by a router. Nowadays, many networks also have firewalls that filter nonsensical and other malicious traffic. In addition, a firewall may be configured to allow only incoming traffic for certain services or connections that are already established. More generally, it is observed that routing and firewalling rules are intrinsically related [16].

In this section, we use constraints in the form of Boolean predicates to describe the type of traffic that is allowed for the ports of a flow.

3.1 Defining Constraint System

Let TypeLit range over integers (for port number), symbols for protocol (e.g., tcp, udp, icmp), and IP addresses (in the form $www.xxx.yyy.zzz/nn$ where nn is prefix length). When appearing alone, these type literals do not have a meaning because of ambiguity. For example, when we see an IP address, is it matched against the source or destination IP address of a packet? Similarly, when we see an integer, is it matched against the source or destination port number?

To disambiguate the matching, we introduce five type constructors:

$$\begin{array}{rcl} \nu \in \mathsf{Type}_0 & ::= & t \mid \alpha \\ \tau \in \mathsf{Type} & ::= & \cdots \mid \mathsf{ipaddr}_{src}\nu \mid \mathsf{ipaddr}_{dst}\nu \\ & & \mid \mathsf{port}_{src}[\nu,\nu] \mid \mathsf{port}_{dst}[\nu,\nu] \mid \mathsf{proto}\ \nu \end{array}$$

The presence of two fields in port constructors allows us to specify a range of ports rather than a singleton port; prefix length of an IP address already allows us to specify a range of IP addresses. We also have negation ($\neg$) as unary type constructor, conjunction ($\wedge$) and disjunction ($\vee$) as binary type constructors. The resulting Type lets us specify basic firewalling rules, restricting properties of a packet based on protocol, source and destination IP address, and port number in a packet's header information.

The meaning of subtyping ($<:$) is treated as logical implication ($\Rightarrow$), and equality ($\doteq$) the same as logical equivalence ($\Leftrightarrow$). In addition, we have predicates such as "does not imply" ($\not\Rightarrow$) and "are not equivalent" ($\not\Leftrightarrow$), which obviously are neither partial order nor equivalence relations. A set of constraints $C = \{c_1, \ldots, c_n\}$ is treated as if $c_1 \wedge \cdots \wedge c_n$. However, this does not mean that such set is in conjunctive normal form.

Special considerations about these type constructors are observed as follows:

- Conjunction of disjoint sets of IP addresses and port ranges is equivalent to $false$; source and destination are considered separately.
- Conjunction of conflicting protocols is equivalent to $false$.

Solving satisfiability of Boolean constraints in general is NP-complete. Binary Decision Diagram is a possible constraint solving algorithm, which has worst case exponential space but efficient in practice.

3.2 Predicate Routing

The idea of predicate routing is to unify the specification of firewall and routing rules using binary predicates—as described in the previous subsection—so they can be easily deployed and checked for error [16]. We demonstrate how to specify flow types of different kinds of network nodes, which we assume are well-behaving; that is, they behave according to the predicate routing rules that describe them. Enforcement of these rules belongs to be role of firewall or packet filter. Since $\mathsf{TRAFFIC}(\mathbf{X})$ lacks the aspect of deployment, we focus instead on the verification part.

Static IP Host A well-behaving host produces an IP packet whose source is the IP address of the host. It also only accept packets whose destination IP address is the same IP address as the host. Due to the way $\mathsf{TRAFFIC}(\mathbf{X})$ models a network as an end-to-end bi-directional flow, upstream hosts and downstream hosts are specified differently (but one is the transpose of the other). Assuming the IP address of the upstream host is ip_1, and downstream is ip_2, we write their types as:

$$\mathsf{type}(Upstream) = \begin{bmatrix} F_{\mathsf{nil}} & \mathsf{ipaddr}_{src} ip_1 \\ F_{\mathsf{nil}} & \mathsf{ipaddr}_{dst} ip_1 \end{bmatrix} \quad \mathsf{type}(Downstream) = \begin{bmatrix} \mathsf{ipaddr}_{dst} ip_2 & F_{\mathsf{nil}} \\ \mathsf{ipaddr}_{src} ip_2 & F_{\mathsf{nil}} \end{bmatrix}$$

Dynamic IP (DHCP) Host A host whose IP address is dynamically configured can be expressed in $\mathsf{TRAFFIC}(\mathbf{X})$ using $\forall$-quantifier. Assuming now that upstream host belongs to IP subnet net_1, and downstream to IP subnet net_2, we can write their types as follows. Notice that subnets are simply IP address with a prefix length shorter than the full length (32 bit).

$$\begin{array}{rcl} \mathsf{type}(Upstream) & = & \forall\alpha \left(\begin{array}{l} \mathsf{ipaddr}_{src}\alpha \wedge (\mathsf{ipaddr}_{src}\alpha \Rightarrow \mathsf{ipaddr}_{src} net_1) \\ \mathsf{ipaddr}_{dst}\alpha \wedge (\mathsf{ipaddr}_{dst}\alpha \Rightarrow \mathsf{ipaddr}_{dst} net_1) \end{array} \right) \cdot \begin{bmatrix} F_{\mathsf{nil}} & \mathsf{ipaddr}_{src}\alpha \\ F_{\mathsf{nil}} & \mathsf{ipaddr}_{dst}\alpha \end{bmatrix} \\ \mathsf{type}(Downstream) & = & \forall\alpha \left(\begin{array}{l} \mathsf{ipaddr}_{src}\alpha \wedge (\mathsf{ipaddr}_{src}\alpha \Rightarrow \mathsf{ipaddr}_{src} net_2) \\ \mathsf{ipaddr}_{src}\alpha \wedge (\mathsf{ipaddr}_{dst}\alpha \Rightarrow \mathsf{ipaddr}_{dst} net_2) \end{array} \right) \cdot \begin{bmatrix} \mathsf{ipaddr}_{dst}\alpha & F_{\mathsf{nil}} \\ \mathsf{ipaddr}_{src}\alpha & F_{\mathsf{nil}} \end{bmatrix} \end{array}$$

The constraint $(\mathsf{ipaddr}_?\alpha) \wedge (\mathsf{ipaddr}_?\alpha \Rightarrow \mathsf{ipaddr}_? net)$ establishes that α must be instantiated to be a subnet of net by *modus ponens*.

Router A router is a device that sits between two networks, relaying traffic from one network to another. Suppose the two networks are net_1 and net_2, we can write its type as:

$$\mathsf{type}(Router) \quad = \quad \begin{bmatrix} \mathsf{ipaddr}_{src} net_1 \wedge \mathsf{ipaddr}_{dst} net_2 & \mathsf{ipaddr}_{src} net_1 \wedge \mathsf{ipaddr}_{dst} net_2 \\ \mathsf{ipaddr}_{src} net_2 \wedge \mathsf{ipaddr}_{dst} net_1 & \mathsf{ipaddr}_{src} net_2 \wedge \mathsf{ipaddr}_{dst} net_1 \end{bmatrix}$$

NAT Device A Network Address Translation device is like a router, typically used to connect between a local area network (LAN) and a wide area network (WAN). Unlike a router, it mangles packet header in the following manner: when a packet from LAN is received, NAT rewrites its source address to make it appear as if it originates from the NAT device. When a packet from WAN arrives in response to the first packet, NAT rewrites its destination address to that of the initiative host. Such translation is transparent to the original source and destination hosts.

NAT uses a state table to keep track of connections in order to rewrite packets forth and back. Without loss of generality, we can write the type as if NAT is tracking only one connection. Assuming the device has a WAN IP address ip assigned statically,

$$\mathsf{type}(NAT) \quad = \quad \forall \alpha, \beta. \begin{bmatrix} \mathsf{ipaddr}_{src} \alpha \wedge \mathsf{ipaddr}_{dst} \beta & \mathsf{ipaddr}_{src} ip \wedge \mathsf{ipaddr}_{dst} \beta \\ \mathsf{ipaddr}_{src} \beta \wedge \mathsf{ipaddr}_{dst} \alpha & \mathsf{ipaddr}_{src} \beta \wedge \mathsf{ipaddr}_{dst} ip \end{bmatrix}$$

In this setting, LAN connects to the left of NAT, and WAN to its right.

Using subtyping by logical implication, $\mathsf{TRAFFIC}(\mathbf{X})$ is able to verify the correctness of firewall rules among the connected components.

4 Conclusion

We described our formal verification framework $\mathsf{TRAFFIC}(\mathbf{X})$ that is specifically tailored for networked application. We proposed two ways to customize the constraint system: using linear inequality constraints, solved by simplex algorithm; and using Boolean predicate constraints, solved by Binary Decision Diagram. We provided several network-centric examples to demonstrate to use of these constraint systems. For these examples, we considered network engineering problems such as maximum transmission unit, timing, bandwidth allocation, firewall and routing, and showed that $\mathsf{TRAFFIC}(\mathbf{X})$ can formally verify these examples with ease.

$\mathsf{TRAFFIC}(\mathbf{X})$ was first described in [8], and this paper extends the motivation of the system. Our future work will focus on implementing:

- A type inference algorithm for $\mathsf{TRAFFIC}(\mathbf{X})$, so flows can be verified without explicit type annotation.
- Constraint solvers for linear inequality constraints and Boolean predicates.
- Constraint solvers using Constraint Handling Rules [5].

and making the system available for the public.

References

[1] Lee Breslau, Deborah Estrin, Kevin R. Fall, Sally Floyd, John S. Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[2] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. *ICCAD*, 00:0236, 1995.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, Cambridge, Mass., 2nd edition, 2001.

[4] Paul Curzon. The formal verification of the Fairisle ATM switching element. Technical Report 329, University of Cambridge Computer Laboratory, March 1994.

[5] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[6] Hazem Hamed, Ehab Al-Shaer, and Will Marrero. Modeling and verification of ipsec and vpn security policies. *icnp*, 0:259–278, 2005.

[7] Institut National de Recherche en Informatiquei et en Automatique INRIA. *The Coq Proof Assistant Reference Manual*, version 8.1 edition, 2006.

[8] Likai Liu and Assaf Kfoury. Safe compositional specification of network systems with polymorphic, constrained types. Technical Report BUCS-TR-2006-029, Computer Science Department, Boston University, Boston, MA, October 25 2006.

[9] Robin Milner. *Communication and mobile systems: the π-calculus*. Cambridge University Press, Cambridge; New York, 1999.

[10] J. C. Mogul and S. E. Deering. Path MTU discovery. RFC 1191 (Draft Standard), Nov 1990.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, Berlin; New York, 2002.

[12] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, Jun 1992. Springer-Verlag.

[13] James Lyle Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[14] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, 1962.

[15] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999.

[16] Timothy Roscoe, Steven Hand, Rebecca Isaacs, Richard Mortier, and Paul W. Jardetzky. Predicate routing: enabling controlled networking. *Computer Communication Review*, 33(1):65–70, 2003.

A The Type System $\mathsf{TRAFFIC}(\mathbf{X})$

A.1 Constraints

It is important to keep in mind that constraints in a set $C = \{c_1, c_2, \ldots c_n\}$ are logically conjunct, i.e., $c_1 \wedge c_2 \wedge \cdots \wedge c_n$, so the union of two sets $C \cup D$ really means $C \wedge D$.

Notation A.1 (Substitution). Let ψ with the sort $\mathsf{TypeVar} \rightarrow \mathsf{Type}$ be a partial mapping that can be applied on types, constraints, flow types, and environments.

Assumption A.2 (Properties of Constraint Entailment)**.** *Given constraint sets C and D, any constraint system that models our notion of constraint entailment $C \Vdash D$ must satisfy these properties:*

$$\textit{(i)}\ \frac{C \Vdash D}{C \cup C' \Vdash D} \qquad \textit{(ii)}\ \frac{C \Vdash D}{\psi C \Vdash \psi D} \qquad \textit{(iii)}\ \frac{C \Vdash D \quad C \cup D \Vdash E}{C \Vdash E}$$

$$\textit{(iv)}\ \frac{D \subseteq C}{C \Vdash D} \qquad \textit{(v)}\ \frac{C \Vdash D \quad C \Vdash E}{C \Vdash D \cup E} \qquad \textit{(vi)}\ \frac{C \Vdash D_1 \cup D_2}{C \Vdash D_i} \quad i = 1, 2$$

A.2 Typing Rules

Notation A.3 (Environment). Let Γ with the sort $\mathsf{FlowVar} \to \mathsf{FlowTypeScheme}$ be a partial mapping, written as:

$$\Gamma = \{x_1 : S_1, \ldots, x_n : S_n\}$$

Notation A.4 (Typing Judgment). A typing judgment, written as $C, \Gamma \vdash \mathcal{A} : S$, means that under the assumption of constraint set C and environment Γ, the flow $\mathcal{A}$ has a flow type scheme S, according to the typing rules in Figure A.1.

$$\text{(VAR-ADD)}\quad \frac{\Gamma(x) = \forall \bar{\alpha}(D).T \quad C \cup D \not\Vdash \mathit{false}}{C \cup D, \Gamma \vdash x : \forall \bar{\alpha}(D).T} \qquad \text{(LET)}\quad \frac{C, \Gamma \vdash \mathcal{A} : S \quad C, \Gamma \cup \{x : S\} \vdash \mathcal{B} : T'}{C, \Gamma \vdash \mathbf{let}\ x = \mathcal{A}\ \mathbf{in}\ \mathcal{B} : T'}$$

$$\text{(LOCAL-ADD)}\quad \frac{\mathsf{type}(A) = \forall \bar{\alpha}(D).T \quad C \cup D \not\Vdash \mathit{false}}{C \cup D, \Gamma \vdash A : \forall \bar{\alpha}(D).T} \qquad \text{(PAR)}\quad \frac{C, \Gamma \vdash \mathcal{A} : \left[\begin{smallmatrix} \tau_1 & \tau_2 \\ \tau_3 & \tau_4 \end{smallmatrix}\right] \quad C, \Gamma \vdash \mathcal{B} : \left[\begin{smallmatrix} \tau_5 & \tau_6 \\ \tau_7 & \tau_8 \end{smallmatrix}\right]}{C, \Gamma \vdash \mathcal{A} \,\|\, \mathcal{B} : \left[\begin{smallmatrix} \tau_1 \cdot \tau_5 & \tau_2 \cdot \tau_6 \\ \tau_3 \cdot \tau_7 & \tau_4 \cdot \tau_8 \end{smallmatrix}\right]}$$

$$\text{(SEQ)}\quad \frac{C, \Gamma \vdash \mathcal{A} : \left[\begin{smallmatrix} \tau_1 & \tau_2 \\ \tau_3 & \tau_4 \end{smallmatrix}\right] \quad C, \Gamma \vdash \mathcal{B} : \left[\begin{smallmatrix} \tau_5 & \tau_6 \\ \tau_7 & \tau_8 \end{smallmatrix}\right] \quad C \Vdash \{\tau_2 \doteq \tau_5, \tau_7 \doteq \tau_4\}}{C, \Gamma \vdash \mathcal{A}; \mathcal{B} : \left[\begin{smallmatrix} \tau_1 & \tau_6 \\ \tau_3 & \tau_8 \end{smallmatrix}\right]}$$

$$\text{(}\forall\text{-INTRO)}\quad \frac{C \cup D, \Gamma \vdash \mathcal{A} : T \quad \bar{\alpha} \cap \mathrm{ftv}(C, \Gamma) = \varnothing}{C \cup D, \Gamma \vdash \mathcal{A} : \forall \bar{\alpha}(D).T} \qquad \text{(SUB)}\quad \frac{C, \Gamma \vdash \mathcal{A} : T \quad C \Vdash T <: T'}{C, \Gamma \vdash \mathcal{A} : T'}$$

$$\text{(}\forall\text{-ELIM)}\quad \frac{C, \Gamma \vdash \mathcal{A} : \forall \bar{\alpha}(D).T \quad C \Vdash \psi D}{C, \Gamma \vdash \mathcal{A} : \psi T} \quad \text{where } \mathrm{dom}(\psi) = \bar{\alpha}.$$

Figure A.1: Typing rules of $\mathsf{TRAFFIC}(\mathbf{X})$.