Typed Abstraction of Complex Network Compositions*

Azer Bestavros, Adam D. Bradley, Assaf J. Kfoury, and Ibrahim Matta
Department of Computer Science
Boston University

Abstract

The heterogeneity and open nature of network systems make analysis of compositions of components quite challenging, making the design and implementation of robust network services largely inaccessible to the average programmer. We propose the development of a novel type system and practical type spaces which reflect simplified representations of the results and conclusions which can be derived from complex compositional theories in more accessible ways, essentially allowing the system architect or programmer to be exposed only to the inputs and output of compositional analysis without having to be familiar with the ins and outs of its internals. Toward this end we present the TRAFFIC (Typed Representation and Analysis of Flows For Interoperability Checks) framework, a simple flow-composition and typing language with corresponding type system. We then discuss and demonstrate the expressive power of a type space for TRAFFIC derived from the network calculus, allowing us to reason about and infer such properties as data arrival, transit, and loss rates in large composite network applications.

1. Introduction

In a recent paper [2], we argued that specifying, designing, and developing correct, efficient, and resilient systems is a notoriously hard problem, particularly when placing these systems in open contexts in which they will interact with dynamic and unpredictable environments, peers, and adversaries. By "correct" we mean that we know with certainty some desirable invariants of a system. Many techniques are already available to describe, discuss, and deduce the invariants of a single software component: type systems, model checking, mathematical analyses and countless derivative tools allow us to speak confidently about many *local invariants* (*e.g.*, well-formed output, minimum throughput, maximum response time, *etc*).

While there are many interesting, useful, and plausibly verifiable properties of single software agents for which plausible verification systems do not yet exist, we do possess a fairly good handle on what invariant properties for single software components *look like* and how to go about expressing and testing them. What we do not yet have is a solid grasp upon how to describe, discuss, and deduce the global invariants of open, extensible software systems, or how to (hopefully efficiently) bridge the gap between local and *global invariants*, where global invariants describe the acceptable range of behaviors and emergent properties when the components or agents making up a system interact (*e.g.*, deadlock-free, stable, fair, *etc*).

This paper presents TRAFFIC, our prototype of a specification language and type inference model that is able to take results and contributions from valuable but less accessible approaches to compositional verification and mechanically integrate their results to check for potential problems in a given system or design. TRAFFIC builds upon a large body of work on software development paradigms for networked systems (*e.g.*, [12, 8, 7]) and formalisms for the analysis of their behaviors (*e.g.*, [4, 10, 9, 11, 6])[1] by offering a level of abstraction between the expressive power of those (and other) tools and the programmer's employment of them.

2. Networks of Typed Gadgets

Formal systems like control or scheduling theory allow us to abstract away some properties of a system to make it possible to reason about the system at a level which is impractical while retaining full detail. Sometimes, however, even these abstract representations afford us more detail than we actually require; in such cases, it may be advantageous to utilize "loose" descriptions of systems and components where those less precise descriptions are sufficient to demonstrate some desirable invariants. For example, it may suffice simply to know whether or not a controller is over-damped, whether the aggregate signaling path delay exceeds some threshold, whether the total steady-state error decreases exponentially, polynomially, or simply monotonically in time, or some combination of such properties. These simpler abstractions, in turn, may be more suitable for export into other domains which are interested in reasoning about high-level qualitative properties of the com-

*This work was supported in part by NSF grants ANI-0205294, ANI-0095988, ANI-9986397, and EIA-0202067.

[1] Space limitations preclude a more thorough treatment of this body of literature. Interested readers are referred to the extended version of this paper [3].

ponents and their interactions without getting bogged down in the minutiae of the particular analysis techniques used to derive those underlying (more precise) invariants and conclusions.

Such an approach reflects the very essence of a *type* in the programming languages and formal models sense: an abstract description of some aspect of a system which captures interesting invariants (convergence, TCP-friendliness, rate and time bounds) while discarding detail which may hamper analysis (*i.e.*, which is either unnecessary for proving desirable invariants at higher layers of abstraction, or which may make a decision algorithm intractable or even uncomputable).

Simply put, we believe that a more scalable and accessible approach to the checking of systems for correctness is to move away from *internal* models, in which the inner workings of a component are reduced to a technique-specific abstract caricature, toward *external* models, in which the workings of components are caricatured qualitatively and quantitatively at their interfaces. This allows us to separate representation from computation, *i.e.*, to reason about and from conclusions without regard for their supporting analysis. It is with these observations in mind that we propose the TRAFFIC framework (Typed Representation and Analysis of Flows For Interoperability Checks), in which each of the components of a composite system is represented simply as a black box, with all correctness constraints presented at its points of composition (edges) in the form of types.

3. Safely Composing Typed Gadgets

In order to mechanically infer properties and results from types like those suggested above, we must formally define a domain of applications and rigorously specify rules for the construction and inference of types based upon the elements of those applications. This section surveys the TRAFFIC flow-composition language (presented more fully in [2, 1]) and type checking algorithms (presented more fully in [1]) designed around this goal.[2]

3.1. Flows

The TRAFFIC framework represents all systems as *typed flows* which can be composed to produce more such typed flows. A *flow* is an abstraction for a distinct component or subsystem of a networked application or system. A flow may represent something physical (*e.g.*, an Ethernet switch or a multicast router) or logical (*e.g.*, a multi-hop network backbone or a content delivery service), may be

[2]A live demo of the TRAFFIC type engine, including illustrative examples of TRAFFIC "programs" and type spaces, can be found on-line at `http://www.cs.bu.edu/groups/ibench/`. Development and expansion of this demo is ongoing, lead by Likai Liu <liulk@cs.bu.edu> and Yarom Gabay <yarom@cs.bu.edu>.

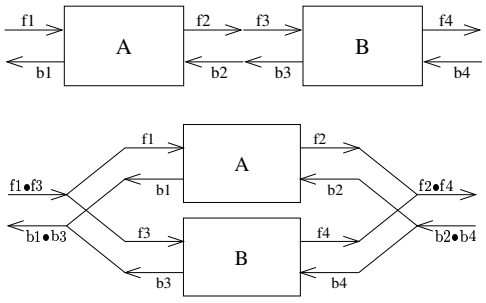

Figure 1. Sequential Composition (top) and Parallel (bottom) Composition.

comprised of several component flows, and may itself be composed with other flows.

We represent flows as boxes with four *sockets* through which they are attached with other flows and by which they are assigned their types. These sockets are directional; two correspond with the "forward" direction (one incoming, one outgoing), two the "backward" direction (one incoming, one outgoing). Intuitively, the flow is a function which operates upon two incoming streams (each guaranteed to adhere to the constraints represented by the *types* of the incoming sockets) and produces two outgoing streams (which are guaranteed to adhere to the constraints represented by the *types* of the outgoing sockets). We attach no particular meaning at this point to "forward" and "backward" apart from their distinctness as such; it may be given additional meaning by the particular type spaces which we will impose upon the sockets below in Section 3.3. In addition, there is little that is particular to having only two channels; this restriction greatly simplifies our syntax and the formal description of our system using familiar language-theoretic tools, but it does not otherwise reflect a fundamental limitations upon the underlying architecture or algorithms.

We now turn to a formal specification of the inner workings of the TRAFFIC framework.

3.2. Specification of Global Flows

We represent a specification of a network application as a *global flow*. A global flow (or simply a *flow*) is a composite object, built up from *local flows* and *flow variables*. Local flows are single components for which we possess complete type information (*e.g.*, A and B in Figure 1(a)); these represent physical or logical intermediaries or endpoints in a specified system. Flow variables are "place holders" in a specification representing components which are not yet known or fully specified; they could represent unknown clients or servers at the endpoints of a specification, unknown intermediary services or transport networks in the middle, or any combinations thereof. In general, global flows can be composed with each other, with local flows, or with flow variables to create larger global flows.

We represent global flows syntactically using the BNF in

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

Figure 2. BNF for Flow Specifications

Figure 2. Intuitively, a *sequential flow* $\mathcal{A}; \mathcal{B}$ is a composition like that in Figure 1(top) in which two flows are placed (logically) adjacently and joined, while a *parallel flow* $\mathcal{A} \| \mathcal{B}$ is one in which two flows are placed (logically) parallel, offering simultaneous rather than serial service and data flow (see Figure 1(bottom)). The sequential operator ";" and the parallel operator "$\|$" have the same precedence, and both associate to the left.

3.3. Syntax of Types

We represent constraints placed upon the behaviors of any component of the system using *types*. Types can be socket types (forward or backward), plain types (forward or backward), or flow types, where types and flow types comprise structured sets of socket types.

A *socket type* is a description of a single logical "entry" or "exit" point for a flow. The type itself may be drawn from any system we have embedded into a partially-ordered type space; examples include the Network Calculus types presented in Section 4 and the systems suggested in Section 5. For example, in a control-oriented application the forward sockets may be typed to describe the steady-state error of traffic use, whether the controller's adaptation is monotonic (over-damped) or not, or the convergence rate or time, and the backward sockets may describe such properties as cumulative feedback delay or feedback origins.

A *plain type* is an ordered list of socket types, describing a (perhaps composite) socket, *i.e.*, a socket which actually corresponds with entry and exit points to one or more parallel flows (as in Figure 1(b)).

A *flow type* is a 4-tuple representing both the forward and backward entry and exit points to a (perhaps composite) flow. This represents the complete type specification of a component in the flow specification. We will present these tuples graphically as two-by-two matrices, *e.g.*, $\begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}$, such that each element's position in the matrix corresponds with the graphical placement of its respective sockets in Figure 1. Intuitively, this means that ρ_1 represents an incoming socket in the forward direction, ρ_2 the forward outgoing socket, σ_1 the *outgoing* socket in the backward direction and σ_2 the *incoming* backward socket.

The syntax of types, and the metavariables ranging over their different categories, are given by the BNF in Figure 3.

$$
\begin{array}{lll}
r & \in \mathsf{FwSocketType} & \\
s & \in \mathsf{BwSocketType} & \\
t & \in \mathsf{SocketType} & ::= r \mid s \\
\rho & \in \mathsf{FwType} & ::= r \mid r\,\rho \\
\sigma & \in \mathsf{BwType} & ::= s \mid s\,\sigma \\
\tau & \in \mathsf{Type} & ::= \rho \mid \sigma \\
T & \in \mathsf{FlowType} & ::= \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}
\end{array}
$$

Figure 3. Syntax of TRAFFIC Types

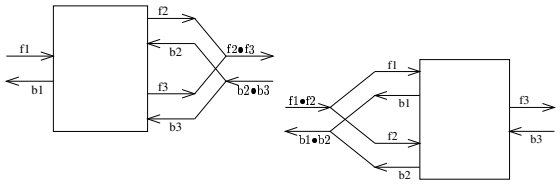

Figure 4. Splitting and Combining Flows

$$
\frac{\{t_1 <: t_2\} \subseteq \Delta}{\Delta \vdash t_1 <: t_2} \qquad \frac{\tau \in \mathsf{Type}}{\Delta \vdash \tau <: \tau} \qquad \frac{T \in \mathsf{FlowType}}{\Delta \vdash T <: T}
$$

$$
\frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta \vdash \tau_2 <: \tau_3}{\Delta \vdash \tau_1 <: \tau_3}
$$

$$
\frac{\Delta \vdash r <: r' \qquad \Delta \vdash \rho <: \rho'}{\Delta \vdash r\,\rho <: r'\,\rho'}
$$

$$
\frac{\Delta \vdash \rho_1' <: \rho_1, \Delta \vdash \rho_2 <: \rho_2', \Delta \vdash \sigma_1 <: \sigma_1', \Delta \vdash \sigma_2' <: \sigma_2}{\Delta \vdash \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} <: \begin{bmatrix} \rho_1' & \rho_2' \\ \sigma_1' & \sigma_2' \end{bmatrix}}
$$

Figure 5. Some TRAFFIC Subtyping rules

3.4. Subtyping

Each potential value for SocketType represents some sort of constraint or invariant upon the qualitative or quantitative behavior of a socket. For most interesting kinds of systems there exist large numbers of such types with strict subset relationships (*i.e.*, we can make many statements of the form "if a socket satisfies a type t_1, then it necessarily also satisfies a less-restrictive type t_2"). We call these relationships *subtypes*, denoting them $t_1 <: t_2$ (read "t_1 is a subtype of t_2" and having the meaning just given). More broadly, where the subtype relationships within a set of SocketType values can be encoded as a partial order, we refer to this set as a *type space*, and refer to the collection of subtype relationships within a type space as Δ. We are then able to extend the subtyping relation to relate flow types using axioms and rules such as those in the illustrative subset presented in Figure 5. Intuitively, these rules are simply establish the reflexivity of subtypes and offer mechanisms for extending subtype relationships from constituent types to composite types, *i.e.*, socket subtype relations can be composed to form plain subtype relations and plain subtype relations can be composed to form flow subtype relations.

A useful operation on plain and flow types is concate-

nation, denoted "•", defined in the obvious way: if $\tau_1 = t_1 \cdots t_m$ and $\tau_2 = t_{m+1} \cdots t_n$ with $m, n \geqslant 1$, then $\tau_1 \bullet \tau_2 = t_1 \cdots t_n$, and extended to flow types:

$$\begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix} \bullet \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix} = \begin{bmatrix} \rho_1 \bullet \rho_3 & \rho_2 \bullet \rho_4 \\ \sigma_1 \bullet \sigma_3 & \sigma_2 \bullet \sigma_4 \end{bmatrix}$$

This allows us to easily construct types for parallel (Figure 1 bottom) and asymmetric (Figure 4) flows.

For example, consider a type system capturing upper and lower bounds upon the value of some property, *e.g.*, $0.3 \leqslant \alpha \leqslant 0.7$. Because the ranges $[0.3, 0.7] \subseteq [0.2, 0.8]$, we say that $(0.3 \leqslant \alpha \leqslant 0.7) <: (0.2 \leqslant \alpha \leqslant 0.8)$ (*i.e.*, if a socket has the former type, it will never be incorrect to treat it as having the latter), or similarly that $(0.2 \leqslant \alpha \leqslant 0.9) <: (0.1 \leqslant \alpha \leqslant 1.0)$, *etc*. This single-socket relationship can then be lifted to compare parallel sockets, *e.g.*,

$$((0.3 \leqslant \alpha \leqslant 0.7) \bullet (0.2 \leqslant \alpha \leqslant 0.9)) <: ((0.2 \leqslant \alpha \leqslant 0.8) \bullet (0.1 \leqslant \alpha \leqslant 1.0))$$

Those familiar with formal type systems will recognize that subtyping on flow types (the last rule) is *contravariant* in the two types along the first diagonal and *covariant* in the two types along the second diagonal; if we think of a flow as a function, this corresponds intuitively with the sockets' *input* and *output* roles, respectively. Thus, again using our bounding types,

$$\begin{bmatrix} (0.1 \leqslant \alpha \leqslant 1.0) & (\alpha = 0.9) \\ (0.4 \leqslant \alpha \leqslant 0.6) & (0.2 \leqslant \alpha \leqslant 0.7) \end{bmatrix} <: \begin{bmatrix} (0.1 \leqslant \alpha \leqslant 0.5) & (0.8 \leqslant \alpha \leqslant 0.9) \\ (0.4 \leqslant \alpha \leqslant 0.6) & (0.4 \leqslant \alpha \leqslant 0.5) \end{bmatrix}$$

(meaning any flow with the first type can be safely used in any context requiring a flow of the second type).

3.5. Typing rules

The point of assigning types to flows and their constituent elements is to enable us to abstract away the internals of each component and perform compositional analysis to assess whether the pieces of a global flow specification will be able to interact according to the declared composition structures, *i.e.*, whether the pieces "fit" together, and what "shape" any gaps (flow variables) in the specification might have. This takes two forms: first, we must describe abstract rules which precisely define the notion of a "fit" when composing typed flows; second, we must define tractable algorithms which are able to evaluate (or at least closely approximate) a determination whether a given application indeed plays by these rules.

A few of the more interesting rules for assigning types to composite flows are described, inductively, in Figure 6. We have defined a set of type-checking algorithms which can be run upon TRAFFIC flows to determine whether, for a given flow and a given type system, any of that flow's constituent compositions could potentially cause some safety property to be violated [1].

$$\text{(par)} \quad \frac{\Gamma, \Delta \vdash \mathcal{A} : T \qquad \Gamma, \Delta \vdash \mathcal{B} : T'}{\Gamma, \Delta \vdash \mathcal{A} \| \mathcal{B} : T \bullet T'}$$

$$\text{(seq)} \quad \frac{\Gamma, \Delta \vdash \mathcal{A} : \begin{bmatrix} \rho_1 & \rho_2 \\ \sigma_1 & \sigma_2 \end{bmatrix}, \ \Gamma, \Delta \vdash \mathcal{B} : \begin{bmatrix} \rho_3 & \rho_4 \\ \sigma_3 & \sigma_4 \end{bmatrix}, \quad \Delta \vdash \rho_2 <: \rho_3, \quad \Delta \vdash \sigma_3 <: \sigma_2}{\Gamma, \Delta \vdash \mathcal{A}\,;\mathcal{B} \ : \begin{bmatrix} \rho_1 & \rho_4 \\ \sigma_1 & \sigma_4 \end{bmatrix}}$$

Figure 6. Some TRAFFIC Typing Rules

4. Network Calculus Types

Many conceptual tools exist for the analysis of the correctness of composite systems, or for understanding emergent properties thereof. While it is expected that network programmers would be well versed in the "art" of specifying and implementing specific functionalities, experience seems to indicate it is unrealistic to assume that they can (or should) master the analytical machinery (and underlying theories) that enable the analysis of the composite system in which such functionalities are embedded (*e.g.*, cascading sequences or parallel sets of controller-driven flows).

We do not believe, however, that this precludes the average programmer from benefiting from the power of such analytical tools. Specifically, we envision development and programming environments which incorporate type systems which reflect simplified forms of such tools, to provide system architects and developers with at least a first-pass approximate answer to whether their system will be stable or unstable, convergent or transient, under-provisioned or over-provisioned, *etc*.

In this section, we present a type space which is modeled after the *Network Calculus* [4], a formalism rooted in the min-plus algebra which offers us tools for reasoning about the capacity of and demand upon networks and network components.

Bear in mind that this type space is not itself TRAFFIC *per se*; rather, it is a demonstration of how a powerful analytical tool (the network calculus) can be distilled and codified as a formal type model to enable its embedding and mechanical application within a generic framework, which may then make those results eligible for export into or composition with other system results and constraints which may derive from completely different verification disciplines. Neither does this system seek to increase the expressive or deductive power of the network calculus; on the contrary, to achieve its goals of abstraction and algorithmic tractability, it will in many cases produce less precise results than a "by hand" analysis might.

We also recognize that the type values themselves as we will present them require a working familiarity with the network calculus to be intelligible. As such, we *do not* expect

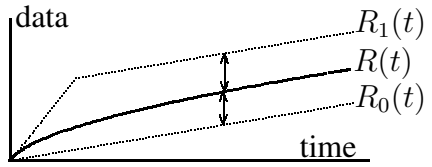

Figure 7. Cumulative flow function R, lower-bounding function R_0 and upper-bounding composite function R_1

for a production prototyping or programming environment to (by default) expose types in this form, but rather expect that system users will interact with a library of parameterized types which describe, in prose and graphical form, the meaning of the invariants to the user in terms familiar to their network engineering (as opposed to mathematical modeling and min-plus algebraic) expertise.

Mathematical Conventions The network calculus uses min-plus algebraic formulas to represent traffic and traffic constraints. All functions in the time domain are defined only over $t \geqslant 0$ with non-decreasing values clamped from below at 0 and are left-continuous or right-continuous.

4.1. Data Flow

In the network calculus, *data flow* is represented by a non-decreasing function R defined over t (time), where $R(t)$ is the number of bits seen at a given point in the system in time interval $[0, t)$. Many other network performance theories focus upon the *rate* at which data is flowing in the system, *i.e.*, $\frac{dR}{dt}$ in this formulation; because the network calculus focuses entirely upon the cumulative function and not upon its derivative (rate), it is not necessary that R even have a derivative (although it is desirable, for purposes of analytical simplicity, that R be at the least either left- or right-continuous).

A particular flow function R is not terribly useful as a class (type). However, we can break the space of all such functions up into regions bounded above or below by other functions. Any non-decreasing, non-negative function R_0 over t can be thought of as defining two "types" of R functions: the type of all functions for which $R(t) \geqslant R_0(t)$ for all t and the type of all functions for which $R(t) \leqslant R_0(t)$ for all t. This is illustrated in Figure 7.

For example, consider the function $f(t) = 0.25t + \sqrt{t}$. All rate functions are trivially bounded from below by the constant function 0, so we say this function is a member of the type $\lfloor\!\lfloor 0 \rfloor\!\rfloor_R$ (where the R subscript identifies this as a type for data flow functions). More interesting, it is also bounded from below by $0.25t$, so it is also a member of the type $\lfloor\!\lfloor 0.25t \rfloor\!\rfloor_R$. It is also bounded from above by (among many others) the affine function $0.75t + 0.5$, and thus belongs to the type $\lceil\!\lceil 0.75t + 0.5 \rceil\!\rceil_R$.

Many of these function shapes have useful analogs in the realm of network monitoring and traffic shaping; for example, affine functions (of the form $At + B$) correspond with the "leaky bucket" traffic shaper, enforcing a long-term steady-state rate limit while allowing for a limited aggregate burst quantity above that rate.

It is particularly interesting to consider the relationships of these bounding functions with each other. Again, think of each function as circumscribing two sets of functions. The set of functions $\lfloor\!\lfloor 0 \rfloor\!\rfloor_R$ is the set of all flow functions; the set $\lfloor\!\lfloor 0.25t \rfloor\!\rfloor_R$ is a subset, so we can say that $\lfloor\!\lfloor 0.25t \rfloor\!\rfloor_R <: \lfloor\!\lfloor 0 \rfloor\!\rfloor_R$. Similarly, $\lceil\!\lceil 0.75t \rceil\!\rceil_R <: \lceil\!\lceil t \rceil\!\rceil_R$, and $\lceil\!\lceil t^2 \rceil\!\rceil_R <: \lceil\!\lceil 2^t + 2 \rceil\!\rceil_R$. In general, for any functions f_0 and f_1, we say that $\lfloor\!\lfloor f_0(t) \rfloor\!\rfloor_R <: \lfloor\!\lfloor f_1(t) \rfloor\!\rfloor_R$ iff $f_1(t) \leqslant f_0(t)$ for all t and that $\lceil\!\lceil f_0(t) \rceil\!\rceil_R <: \lceil\!\lceil f_1(t) \rceil\!\rceil_R$ iff $f_0(t) \leqslant f_1(t)$ for all t. Of course, any function will belong to an infinite number of such sets; this suggests a particularly interesting lattice of types relating the many different kinds of functions (exponential, polynomial, affine) depending upon their parameters (exponents, coefficients, constants).

In practice, it will often be useful to describe functions as subtypes of several other functions at once; consider again Figure 7, particularly R_1, which is composed of two affine functions. We could state the first and second regions as separate functions,

$$R'_1(t) = r'_1 t \qquad \text{and} \qquad R''_1(t) = r''_1 t + b''$$

and say that both $R \in \lceil\!\lceil R'_1(t) \rceil\!\rceil_R$ and $R \in \lceil\!\lceil R''_1(t) \rceil\!\rceil_R$. Thus, we can decompose many region-wise composite convex functions from (usually) discontinuous functions which may be difficult to manipulate directly in the min-plus algebra into collections of simple function types which can be reasoned about individually, having only their results composed. A similar technique is to consider only the *convex hull* of an analytically unmanageable bounding expression, sacrificing some precision for significantly greater ease of representation, manipulation, and analysis.

In either case, what is produced is recognizable to type theorists as an *intersection type*: a given socket's type (in this case, amounts of data arriving at or leaving from a socket) is the intersection of multiple distinct sets (types) which might not themselves have subset/subtype relationships. Intuitively, an expression (socket) must satisfy the conditions of *all* members of the intersection type in order to type-check. We denote this using the $\cap$ operator on types, *e.g.*, $R(t)$ in Figure 7 belongs to the intersection type $\lceil\!\lceil R'_1(t) \rceil\!\rceil_R \cap \lceil\!\lceil R''_1(t) \rceil\!\rceil_R \cap \lfloor\!\lfloor R_0(t) \rfloor\!\rfloor_R$.

We will also make use of the complimentary notion of a *union type*, denoted $\cup$, which means (intuitively) that an expression (socket) may satisfy the conditions of *any* member of the union in order to be deemed correct. We will use this construct to simplify the expression of discontinuous functions which bound a *concave* space. Union and intersection types can (in principle) be mixed, but we do not do so in this paper.

4.2. Arrival Curves

An arrival curve sets an upper bound upon how many bits may be sent within a window of time. Arrival curves are different from the upper-bound data flow functions we discussed above in that the data flow bounds were allowed to be a function of absolute time, while an arrival curve limits how much data can be sent *over a given-length window at any absolute time*; in this sense, it is *sliding window* restriction. Therefore, it is not adequate to show for some arrival curve α and some R that $R(t) \leqslant \alpha(t)$ for all t; an arrival curve imposes the much stronger result that for every t and for all $s \leqslant t$, $R(t) - R(s) \leqslant \alpha(t-s)$.

For example, if R has an arrival curve of $\alpha(t) = rt$, then that flow is rate-limited to r, *i.e.*, over any window of time of any positive length τ, R may send no more than τr bits. Similarly, an affine arrival curve $\alpha(t) = rt + b$ is analogous to the continuous form of the *leaky bucket*, restricting the flow's cumulative deviation from the steady state rate r to the maximum cumulative burst size b.

Taking a hint from our bounding functions above, we can also establish classes (types) of arrival curves, realizing all of the same benefits; specifically, if some function f is an upper-bound of function f', *i.e.*, $f(t) \geqslant f'(t)$ for all t, then we say that $\lceil\!\lceil f' \rceil\!\rceil_\alpha <: \lceil\!\lceil f \rceil\!\rceil_\alpha$, and that $\lfloor\!\lfloor f \rfloor\!\rfloor_\alpha <: \lfloor\!\lfloor f' \rfloor\!\rfloor_\alpha$ (where the α subscripts indicate that these are types of arrival curves). Note that because arrival curves are themselves upper-bounds, lower-bound types have very limited use ("the arrival process is no more than some function which is no less than..."); in practice, we will usually see and use upper-bound arrival curve types.

More interestingly, we can draw relations between arrival curves and data flow functions, namely, whether some data flow function R is agreeable to some arrival curve α. Ideally, we would like to encode this as subtype relationships; for example, we can say for any f that $\lceil\!\lceil f \rceil\!\rceil_\alpha <: \lceil\!\lceil f \rceil\!\rceil_R$ since any R process with an arrival curve α is certainly constrained by the window beginning at absolute time 0. (It is actually more tightly constrained than that, but such is the point of a subtype-supertype relationship: we are willing to shed some detail about the constraint in order to produce a still-valid but less-precise restraint in a different form, which may be more useful.) A slightly less-obvious family of such relationships has the form $\lceil\!\lceil f \rceil\!\rceil_R <: \lfloor\!\lfloor g \rfloor\!\rfloor_\alpha$, meaning that the data flow of *at most* f "fits" within an arrival curve of *at least* g, which holds (as stated above) iff $f(t) - f(s) \leqslant g(t-s)$. There are many obvious examples of such functions, *e.g.*, $\lceil\!\lceil \sqrt{t} \rceil\!\rceil_R <: \lfloor\!\lfloor t+1 \rfloor\!\rfloor_\alpha$ (square-root grows sub-linearly after an initial burst), or (perhaps more interestingly), for any positive-sloped affine function $rt+b$, $\lceil\!\lceil rt+b \rceil\!\rceil_R <: \lfloor\!\lfloor rt+b \rfloor\!\rfloor_\alpha$ (if the total transmission volume in absolute time is leaky-bucket-limited, then marginal transmission volume cannot be limited by anything stricter than the same leaky bucket).

While not defined explicitly in [4], we define a complimentary lower-bounding service curve $\bar{\alpha}$ which expresses the least amount of data that can be sent over a given-length window at any absolute time. For much the same reasons as given above, these will generally only be manifested as lower-bounding types, *i.e.*, having the form $\lfloor\!\lfloor f \rfloor\!\rfloor_{\bar{\alpha}}$.

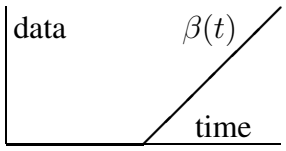

Figure 8. Example service curve $\beta(t)$

4.3. Service Curves and Shapers

Consider a flow with a known incoming data flow curve of R and an unknown outgoing data flow curve R^*. The network calculus defines the *service curve*, denoted as β, as a way of relating these two: given a particular R, the service curve gives us a way of defining a worst-case bound upon what R^* can be. Specifically, for all t,

$$R^*(t) \geqslant \min_{s \leqslant t} \left(R(s) + \beta(t-s) \right)$$

Intuitively, β is a guarantee that, over any window of size t, a minimum of $\beta(t)$ bits can be conducted across the service. Thus, the input-output constraint looks (for any t) at the arrival history (R over $0 \leqslant t$) and finds the worst ($\min$) possible window ($t - s$ where $s \leqslant t$) that could limit the number of bits transiting the component ($R(t_0) + \beta(t - t_0)$ for some past t_0). Often, service curves will have shapes like that shown in Figure 8; intuitively, this particular shape indicates that the user/programmer must allow for some initial latency (the flat region beginning at $t = 0$), after which the sending rate is governed by a steady-state limit (the slope of the later region).

In the min-plus algebra, upon which the network calculus is based, this is more simply stated as a convolution: $R^* \geqslant R \otimes \beta$. Stated using our model of bounding function constraint types, a service curve β on an outgoing socket implies that, if the corresponding incoming socket has type $\lfloor\!\lfloor F \rfloor\!\rfloor_R$, then the output socket has the type $\lfloor\!\lfloor F \otimes \beta \rfloor\!\rfloor_R$.

Analogous to the service curve is a *shaper*, denoted σ, which defines a best-case characterization of how traffic may transit a node. Intuitively, the shaper function is a sliding-window constraint which will cause the transmission of any data exceeding its shape to be delayed through the use of a (potentially unlimited) buffer. (We present a basic primitive for dealing with finite-sized buffers below.) If we have an upper-bound upon a shaper $\lceil\!\lceil f \rceil\!\rceil_\sigma$ and an upper bound upon an input function $\lceil\!\lceil g \rceil\!\rceil_R$, we then also have an upper bound upon the output, $\lceil\!\lceil f \otimes g \rceil\!\rceil_R$.

4.4. Losses

The network calculus can also represent losses in the network driven by *capacity* limits (*i.e.*, buffer overflows).

Specifically, for some node with an arrival curve $\alpha(t)$, we say there is a corresponding *admission curve* $x(t)$ which reflects cumulative data not rejected from the node because of a capacity constraint over any internal t. We say $L(t)$ represents the total cumulative rejected data over interval t, *i.e.*, $\alpha(t) - x(t)$, and that $l(t)$ is the cumulative loss rate per unit of data (*i.e.*, $L(t)/\alpha(t)$). As above, we can express upper and lower bounds upon cumulative loss and loss rates using our bounding function notation and the L and l subscripts, respectively; for example, a flow experiencing a loss rate of no more than 5 units data per unit time would have the type $\lceil 5t \rceil_L$ while one experiencing at least 1% loss would have the type $\lfloor 0.01 \rfloor_l$.

Loss rates provide us with a mechanism for limiting best-case service; for example, given input flow $R(t)$ and losses $L(t)$, we trivially infer that the output $R^*(t)$ is constrained to $\lceil R(t) - L(t) \rceil_R$, or (more generally) that if losses are of type $\lfloor Q(t) \rfloor_L$ (best-case minimum loss regime of Q), then

$$R^*(t) \in \lceil R(t) - Q(t) \rceil_R,$$

and that if losses are of the type $\lceil Q(t) \rceil_L$ (worst-case maximal loss regime of Q), then

$$R^*(t) \in \lfloor (R(t) - Q(t)) \otimes \beta \rfloor_R.$$

4.5. Bringing it Together

The fact that service curves define lower bounds upon data flow functions fits very nicely with our notion of bounding functions as types. Consider a gadget which queues, schedules, and forwards packets in its forward path and passes control data on its backward path. Suppose this gadget has a limiting arrival curve (A), a minimal arrival rate ($\lfloor R_0 \rfloor_R$), and its traffic-shaping behavior is characterized by a service curve (call it B). With only this information and a simple type system, we can specify a generic type signature for this gadget:

$$\begin{bmatrix} \lceil A \rceil_\alpha \cap \lfloor R_0 \rfloor_R & \lceil A \rceil_\alpha \cap \lfloor R_0 \otimes B \rfloor_R \\ \textit{control} & \textit{control} \end{bmatrix}$$

where $\cap$ denotes an *intersection type*, *i.e.*, for an expression to conform to type $A \cap B$ it must conform both to A and to B.

However, if we expect to employ this gadget in a wide variety of settings, the bounds of $\lceil A \rceil_\alpha$ and $\lfloor R_0 \rfloor_R$ may be so permissive as to discard most of the expressiveness the network calculus affords us. However, if we introduce min-plus convolution as an operator upon types, we can re-state the above gadget's type *polymorphically* as

$$\forall x. \begin{bmatrix} \lceil A \rceil_\alpha \cap \lfloor x \rfloor_R & \lceil A \rceil_\alpha \cap \lfloor x \otimes B \rfloor_R \\ \textit{control} & \textit{control} \end{bmatrix}$$

This allows the lower-bound type for the forward output to be dependent upon the lower-bound constraints upon the input; thus, if we have a precise lower-bound at the input, we are able to derive an equally precise lower-bound at the output, and if we are presented only with an approximate (but correct) lower bound at the input, we are able to derive a similarly approximate (but still correct) lower-bound upon the output.

More generally speaking, there exist many such relational dependencies between the varieties of types we have defined, and each such relationship can be thought of as defining an implicit polymorphic type which can be applied to any flow. Consider again the above example: a flow's outgoing rate is bounded from below by the convolution of the incoming rate's lower-bound with the service curve. Notice that, given any two of these types, we can algebraically derive the third. More generally, we have defined the following implicit "type templates" which allow us to do this with various combinations of upper and lower bounding R, σ, and L types:

Incoming	Outgoing
$\lfloor F \rfloor_R$	$\lfloor F \otimes \beta \rfloor_R$
$\lceil F \rceil_R$	$\lceil G \rceil_\sigma \cap \lceil F \otimes G \rceil_R$
$\lceil F \rceil_R$	$\lfloor Q \rfloor_L \cap \lceil F - Q \rceil_R$
$\lfloor F \rfloor_R$	$\lceil Q \rceil_L \cap \lfloor (F - Q) \otimes \beta \rfloor_R$
$\lceil F \rceil_R$	$\lfloor Q \rfloor_L \cap \lceil G \rceil_\sigma \cap \lceil (F - Q) \otimes G \rceil_R$

We have also defined several classes of subtyping relationships, such as those between certain data flow functions and certain arrival curves (*e.g.*, $\lceil rt + b \rceil_R <: \lfloor rt + b \rfloor_\alpha$); this relation allows us to transform an existing outgoing data flow bound into an outgoing "arrival curve" bound (*i.e.*, the arrival curve which will hold for the recipient of the outbound stream), and similarly, to convert arrival curves on an inbound socket into data flow curves.

Derivative Types and their Practical Uses In like manner, we are able to (with relative ease) derive (using the min-plus algebra) upper and lower bounds upon a range of interesting and important properties which are *internal* to system components and *derivative of* the properties we have thus far discussed.

For example, queue length (*i.e.*, *backlog*) in a lossless system for any concrete $R(t)$ and $R^*(t)$ is simply $R(t) - R^*(t)$. Given a flow with input type $\lfloor w(t) \rfloor_R \cap \lceil x(t) \rceil_R$ and output type $\lfloor y(t) \rfloor_R \cap \lceil z(t) \rceil_R$, buffer utilization ($B$) is bounded by $\lfloor w(t) - z(t) \rfloor_B \cap \lceil x(t) - y(t) \rceil_B$. In a lossy component where loss is bounded by $\lceil v(t) \rceil_L$, buffering is more tightly bounded above by $\lceil x(t) - (v(t) + y(t)) \rceil_B$.

Thus, we are able to provision our buffer to tolerate bursts up to duration t by simply finding the maximal value of this function over the interval $(0, t]$.

Upper bounds upon delay (D) can be similarly derived as: $\lceil max_{\forall s}(min_{\tau \geqslant 0}(\tau | \alpha(s) \leqslant \beta(s + \tau))) \rceil_D$.

Computation Within Types While many conventional type system require only trivial computation upon the values of the types themselves (specifically, equality and comparison with respect to a partial-order), the type space proposed in this section calls for significant computation to be

performed upon the values of the types themselves as types are inferred/derived for composite flows. Specifically, we have introduced five min-plus algebraic operations: addition (denoted by $+$), subtraction (denoted by $-$), convolution (denoted by $\otimes$), supremum (denoted as $\max$ for intuitive accessibility), and infimum (denotes as $\min$ for the same reason), all of which may be present in a flow's type signature. The presence of these operators and our desire to uncover and capitalize upon subtype relationships between terms comprised of them demands that our type system include notions of algebraic *normalization*, *reduction*, and *comparison* which can be performed upon such terms. We have specified a simple rule set which is capable of performing limited normalization, reduction, and comparison between polynomial terms (omitted here for want of space), and are confident that we can extend this reasoning system to cover a usable subset of the min-plus algebra's representational power (including distributive cases, which occur in the example below) and thereby support our goal of automating network calculus-related computation within the type system rather than exposing it to the user.

4.6. Practica

As we have already noted, this raw presentation of network calculus type forms may not be the appropriate *interface* to present to architects and programmers. It would instead be reasonable to offer to them a library of parameterized templates for these kinds of constraints, described in terms they could readily understand (*e.g.*, "constant rate r", "leaky bucket with rate r and burst limit b", "service at maximum rate r with maximum lag d", *etc*), which are translated "under the hood" to and from the formal network calculus types described here.

4.7. Example Application

A wireless service provider wishes to provide a video aggregation, distillation, and delivery service to two clients, described in the TRAFFIC syntax as

$$\textit{video1} \| \textit{video2}; \textit{shaper}; \textit{delivery}; (\textit{clientA} \| \textit{clientB})$$

and illustrated (with type annotations) in Figure 9. In the interest of space, we will only examine types in the forward direction. The *a priori* design constraints are those denoted by types in the diagram, *i.e.*:

video nodes Each outgoing socket is $\lfloor\!\lfloor t-5 \rfloor\!\rfloor_R \cap \lceil\!\lceil t+5 \rceil\!\rceil_\alpha$. Each video source is variable-bit-rate with a steady-state rate of 1 and burst magnitude of no more than 5.

shaper node Incoming socket is $\lfloor\!\lfloor 2t-10 \rfloor\!\rfloor_R \cap \lceil\!\lceil 2t+10 \rceil\!\rceil_\alpha$. Service curve is $\lfloor\!\lfloor 2t-10 \rfloor\!\rfloor_\beta$ (the node will introduce no more than 5 seconds of total delay), shaper is $\lceil\!\lceil 2t \rceil\!\rceil_\sigma$ (smoothing all transmissions which exceed the steady-state rate of $2t$).

wireless delivery network Loss rate is $\lceil\!\lceil 0.15t+1 \rceil\!\rceil \cap \lfloor\!\lfloor 0.05t \rfloor\!\rfloor$, *i.e.*, the network drops at least a constant, uniform 2.5% of the steady-state stream rate ($2t$) and at most 7.5% plus a single loss burst of 1 unit.

client nodes Each incoming socket is $\lfloor\!\lfloor 1.2t-16 \rfloor\!\rfloor_{\bar{\alpha}} \cup \lfloor\!\lfloor 0.7t-4.5 \rfloor\!\rfloor_{\bar{\alpha}}$, *i.e.*, the client is willing to tolerate a worst-case 30% steady-state loss rate ($0.7t$), a delay of less than 7 seconds in steady state ($0.7t-4.5$), and delay bursts of up to 27 seconds provided they are recovered from at no less than 1.2 times the original transmission rate ($1.2t-16$).

Those sockets about which no type information is established *a priori* (including all backward channel sockets) are denoted by the type variables $\tau_{x,i}$ for $x \in \{a,b,c,d,e,f\}$ and $i \in \{1,2,3,4\}$. Our goal, in part, is to find correct types to assign to these variables in order to determine whether this application will "fit" together as specified.

We will type-check this system using a greedy left-to-right approach, inferring absent α, $\bar{\alpha}$, and R types for each $\tau_{x,1}$ and $\tau_{x,2}$ as we go. Bear in mind, the TRAFFIC compiler would be performing all of these steps automatically, including many inferences not presented here (in the interest of brevity and clarity). Since the number of varieties of types is finite and there are no mechanisms by which the number of members of an intersection type can grow more than linearly, the process is algorithmically no more than low-order polynomial in the number of inference steps (*i.e.*, min-plus algebra normalizations).

First, we check the composite output of the two video nodes against the required input type to the shaper node. The composed output type, precisely, is $\lfloor\!\lfloor t-5 \rfloor\!\rfloor_R \cap \lceil\!\lceil t+5 \rceil\!\rceil_\alpha \bullet \lfloor\!\lfloor t-5 \rfloor\!\rfloor_R \cap \lceil\!\lceil t+5 \rceil\!\rceil_\alpha$; our system must include a primitive for the summation of traffic over a link, which takes the form of simple min-plus addition for both $\lfloor\!\lfloor \cdots \rfloor\!\rfloor_R$ types (lower bounds in absolute time) and $\lceil\!\lceil \cdots \rceil\!\rceil_\alpha$ types (upper bounds on upper bounds over sliding windows); thus, the actual incoming type is $\lfloor\!\lfloor 2t-10 \rfloor\!\rfloor_R \cap \lceil\!\lceil 2t+10 \rceil\!\rceil_\alpha$, which is a subtype of the shaper's input socket because subtyping is reflexive; thus, the first set of sockets type-check.

Second, we determine the output type of the shaper, which is also the input type to the delivery network. Because of the $\lceil\!\lceil 2t \rceil\!\rceil_\sigma$ shaper type, the summed output of the shaper is limited by $\lceil\!\lceil 2t \rceil\!\rceil_\alpha$. However, since we need to maintain the integrity of our type system with respect to each individual channel in order to check the final composition (the two client nodes), we must determine the best-case for each channel, which is $\lceil\!\lceil t+5 \rceil\!\rceil_\alpha \cap \lceil\!\lceil 2t \rceil\!\rceil_\alpha$ for each (intuitively, the 5-unit burst can only build up at a rate of $2t$).

The lower bound is more subtle than it may seem, as the service curve is only given for the *aggregate* flow and not for the *constituent* flows; as such, we are not able to "divide up" the worst-case service delay between the two flows, since either one could (in the worst case) be subject

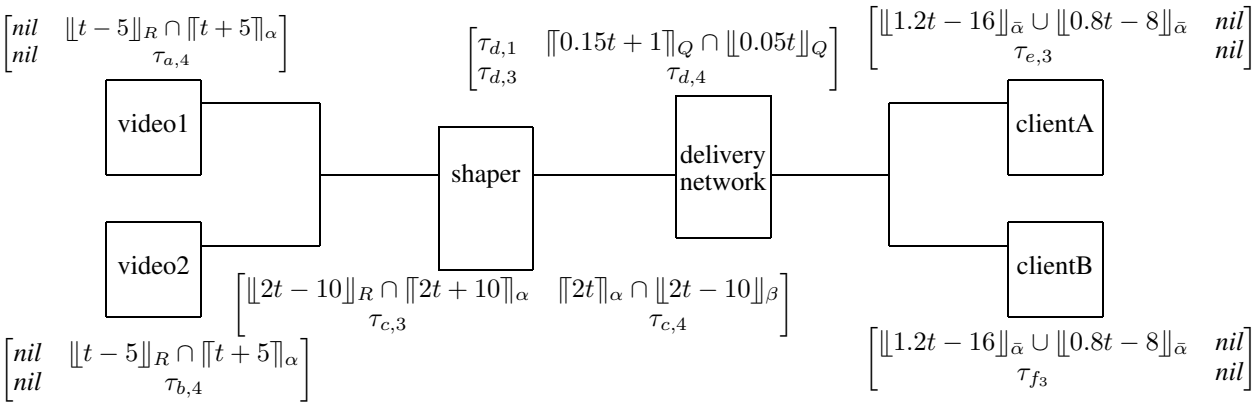

Figure 9. Example TRAFFIC/Network Calculus Application

to the maximal delay while the other is passed through instantaneously. Thus, the output on each channel is $\lfloor\!\lfloor (t-5) \otimes (2t-10) \rfloor\!\rfloor_R$, which (perhaps surprisingly) normalizes to $\lfloor\!\lfloor t-5 \rfloor\!\rfloor_R$, and so the complete outgoing type from the shaper (and unrestrained incoming type to the delivery network) is $\lceil\!\lceil t+5 \rceil\!\rceil_\alpha \cap \lceil\!\lceil 2t \rceil\!\rceil_\alpha \cap \lfloor\!\lfloor t-5 \rfloor\!\rfloor_R \bullet \lceil\!\lceil t+5 \rceil\!\rceil_\alpha \cap \lceil\!\lceil 2t \rceil\!\rceil_\alpha \cap \lfloor\!\lfloor t-5 \rfloor\!\rfloor_R$.

Notice that if we sum the channels, we get $\lfloor\!\lfloor 2t-10 \rfloor\!\rfloor_R$, whereas had we used a summed input type, we would have derived a looser lower bound of $\lfloor\!\lfloor 2t-20 \rfloor\!\rfloor_R$. This is an example of the expression/simplicity tradeoff discussed earlier; by retaining a more detailed (two-channel) type, we are able to more tightly bound the behavior of the system, but by shedding detail (summing the channels) we do not sacrifice the *correctness* of our results but merely their *precision*.

Third, we need to determine the output type of the delivery network based upon its input type, loss type, and service curve type, using the equation we derived above. Since no service or shaping curve is provided, we can substitute any function not less than the input's upper R bound for β, *e.g.*, $\lfloor\!\lfloor 2t+10 \rfloor\!\rfloor_\beta$; min-plus convolution of all such R bounds with β then become identity operations.

Since we have both upper and lower input bounds and upper and lower loss bounds, we can compute both upper and lower output bounds. We cannot directly relate α input and output bounds using losses, so we convert our incoming α upper-bounds (which are affine) directly into (weaker) R upper-bounds. We then use our template relating incoming R upper-bounds with outgoing L lower-bounds to determine the outgoing R upper-bound, namely, $\lceil\!\lceil (t+5) - 0.05t \rceil\!\rceil_R \cap \lceil\!\lceil (2t) - 0.05t \rceil\!\rceil_R$ on each channel, which normalizes to $\lceil\!\lceil 0.95t+5 \rceil\!\rceil_R \cap \lceil\!\lceil 1.95t \rceil\!\rceil_R$.

Similarly, using the template for output lower bound based upon input lower bound, loss rate upper bound, and service curve, we find the output lower bound type on each channel to be $\lfloor\!\lfloor ((t-5) - (0.15t+1)) \otimes 2t \rfloor\!\rfloor_R$ which is normalized easily to $\lfloor\!\lfloor 0.85t-6 \rfloor\!\rfloor_R$. Thus, the output type of the delivery network is $\lceil\!\lceil 0.95t+5 \rceil\!\rceil_R \cap \lceil\!\lceil 1.95t \rceil\!\rceil_R \cap \lfloor\!\lfloor 0.85t-6 \rfloor\!\rfloor_R \bullet \lceil\!\lceil 0.95t+5 \rceil\!\rceil_R \cap \lceil\!\lceil 1.95t \rceil\!\rceil_R \cap \lfloor\!\lfloor 0.85t-6 \rfloor\!\rfloor_R$.

Finally, we make sure the delivery network's output type is compatible with the input types of the clients, *i.e.*, whether $\lceil\!\lceil 0.95t+5 \rceil\!\rceil_R \cap \lceil\!\lceil 1.95t \rceil\!\rceil_R \cap \lfloor\!\lfloor 0.85t-6 \rfloor\!\rfloor_R <: \lfloor\!\lfloor 1.2t-16 \rfloor\!\rfloor_{\bar{\alpha}} \cup \lfloor\!\lfloor 0.7t-4.5 \rfloor\!\rfloor_{\bar{\alpha}}$.

Notice that there is no upper bound on incoming rate or arrival curve; therefore, we disregard the R upper-bound types and must simply assess $\lfloor\!\lfloor 0.85t-6 \rfloor\!\rfloor_R <: \lfloor\!\lfloor 1.2t-16 \rfloor\!\rfloor_{\bar{\alpha}} \cup \lfloor\!\lfloor 0.7t-4.5 \rfloor\!\rfloor_{\bar{\alpha}}$. The type $\lfloor\!\lfloor 0.7t-4.5 \rfloor\!\rfloor_{\bar{\alpha}}$ is itself a supertype of $\lfloor\!\lfloor 0.7t-4.5 \rfloor\!\rfloor_R$; however, $0.85t-6 \not\geqslant 0.8t-8$ for all $t \geqslant 0$ (they have a positive crossing at $t = 10$); using conventional back-of-the-envelope analysis without the benefit of the refined models of the network calculus, this could flag the application as unstable, when in fact we can prove its correctness by ensuring that $\lfloor\!\lfloor 0.85t-6 \rfloor\!\rfloor_R <: \lfloor\!\lfloor 1.2t-16 \rfloor\!\rfloor_{\bar{\alpha}}$ for $t \leqslant 10$, which does in fact hold (by similar analysis). Since then $\lfloor\!\lfloor 0.85t-6 \rfloor\!\rfloor_R <: \lfloor\!\lfloor 1.2t-16 \rfloor\!\rfloor_{\bar{\alpha}} \cup \lfloor\!\lfloor 0.8t-8 \rfloor\!\rfloor_{\bar{\alpha}}$, the composition does type-check beginning to end, and the entire verification process is driven by a mechanical algorithm invisible to the programmer specifying the application.

5. Conclusion and Future Work

The TRAFFIC language and type inference model is in no way particular to the Network Calculus type system we have presented. In addition to our ongoing work developing the core TRAFFIC compiler and type-checking engines and refining and expanding the automated power of our network calculus-based type space, we are exploring several other novel type systems which reflect results in other approaches to compositional analysis.

Scheduling Theory The recent work of Shin and Lee [13] is an example of a system which yields abstract descriptions of the properties of complex components. Rather than working directly with the internal details of real-time scheduling systems (workloads, algorithms, and actual schedules), their periodic resource model affords a straightforward expression of the needs and capabilities of real-time schedulers in terms of a small set of linear equations.

Schedulability is then expressed in terms of necessary and sufficient conditions upon these equations, allowing us to completely set aside all internal details of the system while retaining the ability to precisely determine whether a set of schedulers can be composed under a super-scheduler in a such a way that they will still guarantee their deadlines will be met. Notice also that many expressions of periodic schedules lend themselves very naturally to subtype relationships, *e.g.*, a process able to produce no fewer than a units of output every b units of time is also able to produce na output per nb time for any integer $n \geqslant 1$.

Queuing Theory The analysis of composite systems is common in queuing theory; sequences of queues in open and closed loops are commonly used to describe a whole range of environments and systems. Queuing system descriptions are themselves a kind of type; "$M/M/3/20/100/FIFO$" is a queue taking a memoryless (Poisson) input, memoryless (exponential) service times, three parallel servers, a maximum queue length of 20, a maximum population in the system of 100, and a first-in first-out scheduling discipline. This nomenclature lends itself naturally to defining subtype relationships (*e.g.*, "$M/M/1$" $<:$ "$M/G/1$" $<:$ "$G/G/k$").

Control Theory We have discussed the foundations for a control-theoretic type space in our recent paper [2]. This is another example of a type space which will require extensive support for algebraic reduction and normalization, and may require support for calculus-level operations as well to render it sufficiently powerful to be useful.

Language Theory Issues In addition to our use of involved (albeit still Turing-incomplete) computation systems within the values of types themselves, this research agenda suggests several interesting and promising lines of programming language research. Recognizing from a program's syntax its characterization in terms of compositional theories, *e.g.*, a congestion control function as a **PI** controller or an imperative routine as operationally idempotent [5], is a subtle but not not unsolvable problem, in practice if not in general. Such a capability, even if limited, bridges the gap between the TRAFFIC model of "programming" as the composition of pre-existing, pre-typed "widgets" of network functionality and the actual reliable implementation and typing of the widgets themselves for use in a TRAFFIC-like composition environment.

Another research direction is to extend our specification language to allow simultaneous specification of large collections of desirable network flows. One approach, currently under investigation, involves multiple-choice let-bindings of the following form (*cf.* Figure 2):

$$\textbf{let } x \in \{\mathcal{A}_1, \ldots, \mathcal{A}_n\} \textbf{ in } \mathcal{B}$$

The interpretation (or *semantics*) of such an expression can vary, again depending on the application, while an appropriate typing depends upon the choice of the semantics. The simplest interpretation requires that, for every $\mathcal{A}_i$, the expression "**let** $x = \mathcal{A}_i$ **in** $\mathcal{B}$" will type-check. *Nested* multiple-choice let-bindings would make a brute force approach to testing this claim grow exponentially in cost; this motivates the development of more computationally efficient type-checking algorithms which are commensurately more conservative, *i.e.*, they exchange computational ease for expressive precision (and in so doing, may become incapable of approving some subset of correct global flows).

References

[1] A. Bestavros, A. D. Bradley, A. J. Kfoury, L. Liu, and I. Matta. Type-checking of compositional network flows. Technical Report BUCS-TR-2005-014, Boston University Computer Science, May 2005.

[2] A. Bestavros, A. D. Bradley, A. J. Kfoury, and I. Matta. Safe compositional specification of networking systems. *ACM Computer Communications Review*, 34(3):21–33, 2004.

[3] A. Bestavros, A. D. Bradley, A. J. Kfoury, and I. Matta. Typed abstraction of complex network compositions (extended version). Technical Report BUCS-TR-2005-015, Boston University Computer Science, May 2005.

[4] J.-Y. L. Boudec and P. Thiran. *Network Calculus*. Springer Verlag, May 2004. LNCS 2050.

[5] A. D. Bradley. *A Type-Disciplined Approach to Developing Resources and Applications for the World-Wide Web*. PhD thesis, Boston University, 2004.

[6] J. A. Cobb and M. G. Gouda. Flow theory. *IEEE/ACM Transactions on Networking*, 5(5):661–674, Oct. 1997.

[7] L. Gong. Project JXTA: A technology overview. Technical report, Sun Microsystems, 2000.

[8] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *Computer Networks*, July 2000.

[9] G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. In *Proc. PSTV/FORTE99 Publ. Kluwer*, pages 481–497, Beijing China, Oct. 1999.

[10] J. Liebeherr, S. Patek, and A. Burchard. A calculus for end-to-end statistical service guarantees. Technical report, University of Virginia, 2001.

[11] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3)(3):219–246, Sept. 1989.

[12] M. Pradella, M. Rossi, D. Mandrioli, and A. Coen-Porisini. A formal approach for designing CORBA based applications. In *ICSE 2000*, 2000.

[13] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *24th IEEE International Real-Time Systems Symposium (RTSS'03)*, 2003.