Safe Compositional Network Sketches: Tool & Use Cases

Azer Bestavros
Computer Science Dept
Boston University, MA
best@cs.bu.edu

Assaf Kfoury
Computer Science Dept
Boston University, MA
kfoury@cs.bu.edu

Andrei Lapets
Computer Science Dept
Boston University, MA
lapets@cs.bu.edu

Michael Ocean
Computer Science Dept
Endicott College, MA
mocean@endicott.edu

***Abstract*—NetSketch is a tool that enables the specification of network-flow applications and the certification of desirable safety properties imposed thereon. NetSketch is conceived to assist system integrators in two types of activities: modeling and design. As a modeling tool, it enables the abstraction of an existing system while retaining sufficient information about it to enable future analysis of safety properties. As a design tool, NetSketch enables the exploration of alternative safe designs as well as the identification of minimal requirements for outsourced subsystems. NetSketch embodies a lightweight formal verification philosophy, whereby the power (but not the heavy machinery) of a rigorous formalism is made accessible to users via a friendly interface. NetSketch does so by exposing tradeoffs between exactness of analysis and scalability, and by combining traditional whole-system analysis with a more flexible compositional analysis. The compositional analysis is based on a strongly-typed Domain-Specific Language (DSL) for describing and reasoning about constrained-flow networks at various levels of sketchiness along with invariants that need to be enforced thereupon. In this paper, we overview NetSketch, highlight the tool's philosophy and salient features, and illustrate how it could be used in applications that include: the management/shaping of traffic flows in a vehicular network (as a proxy for CPS applications) and in a streaming media network (as a proxy for Internet applications). In a companion paper, we define the formal system underlying the operation of NetSketch, in particular the DSL behind NetSketch's user-interface when used in "sketch mode", and prove its soundness relative to appropriately-defined notions of validity.**

***Keywords*-Compositional Analysis; Whole-System Analysis; Type Systems, Domain Specific Languages, Network Flow Applications**

I. Motivation and Scope

Traditionally, the design and implementation of trustworthy systems follows a bottom-up approach, enabling system designers and builders to certify (assert and assess) desirable safety invariants of the entire system as a whole. For example, the development of applications with predictable timing properties necessitates the use of special-purpose, real-time kernels so that timing properties at the application layer (top) could be established through knowledge and/or tweaking of much lower-level kernel details (bottom) such as worst-case context switching times and specific scheduling parameters. While justifiable in some instances, this vertical (bottom-up) approach does not lend itself well to current practices in the assembly of complex, large-scale systems – namely, the integration of various subsystems into a whole by "system integrators" who may not necessarily possess the requisite expertise or knowledge of the internals of the subsystems on which they rely. This alternative horizontal approach has significant merits with respect to scalability and modularity, but at the same time it poses significant challenges with respect to aspects of trustworthiness – namely, certifying that the system as a whole will satisfy specific invariants (*e.g.*, related to safety, security, and timeliness). While it is possible to reason about and/or automatically infer the exact (tight) conditions under which safety constraints are satisfied for small-scale (toy), fully-specified subsystems, the same cannot be expected for large-scale, complex systems. Thus, we recognize in this context three specific challenges that the work we present in this paper aims to mitigate.

Exposing Tradeoffs: The environments and tools used by system integrators must expose the inherent tradeoff between the *exactness* of safety analysis of the underlying subsystems, and the *computational complexity* necessary for automated analysis. For example, it should be possible for a system integrator to *sketch* (under-specify) any guarantees or constraints expected to hold in a subsystem for less critical functionalities (or early on in the design phase), and yet expect a level of support for system-wide safety analysis that is commensurate with the provided details. Such a capability would enable system integrators to establish "minimal" subsystem requirements ensuring system-wide safety properties. Similarily, it should be possible for a system integrator to escalate the automated analysis of safety properties based on the low computational cost of such an analysis.

Lowering the Bar: Support for safety analysis in design and/or development environments must be based on sound formalisms that are not specific to (and do not require expertise in) particular domains. While acceptable and perhaps expected for vertically-designed and smaller-scale (sub)systems, deep domain expertise cannot be assumed for designers of horizontally-integrated, large-scale systems. Not only should the underlying formalism be domain-agnostic, it must also be possible for the formalism to act as a unifying glue across multiple theories and calculi. In particular, such a formalism should enable system integrators to manipulate results obtained through multiple, esoteric domain-specific techniques (*e.g.*, using network calculus to obtain worst-case delay envelopes, using scheduling theory to derive upper bounds on resource utilizations, or using queuing theory to derive steady-state average delays). [1] In doing so, we lower the bar of expertise required to take full advantage of such domain-specific results at the small (subsystem) scale, while at the same time enabling scalability of safety analysis at the large (system) scale.

Enabling Compositional Network Flow Analysis: Most large-scale systems are modeled/viewed as interconnections of subsystems, or *gadgets*, each of which is a producer, consumer, or regulator of *flows* that are characterized by

[1] Naturally, end-user tools that leverage such formalism would be customized to present *entities* relevent to specific application domains.

Gadgets	Whole-system	vs	Compositional analysis
A	$[\![A]\!]$	$=$	$[\![A]\!]$
$A \otimes B \otimes C$	$[\![A \otimes B \otimes C]\!]$	$=$	$[\![A]\!] \star [\![B]\!] \star [\![C]\!]$
$A \otimes \langle _ \rangle \otimes C$	$[\![A \otimes \langle _ \rangle \otimes C]\!]$?	$\stackrel{?}{=}$	$[\![A]\!] \star [\![\langle _ \rangle]\!] \star [\![C]\!]$
$A \otimes B' \otimes C$	$[\![A \otimes B' \otimes C]\!]$	$=$	$[\![A]\!] \star [\![B']\!] \star [\![C]\!]$

Figure 1. Contrasting whole-system and compositional analyses.

a set of variables and a set of constraints thereof, reflecting *inherent* or *assumed* properties or rules for how the gadgets operate (and what constitutes safe operation). We argue that system integration can be seen primarily as a *network flow* management exercise, and consequently that tools developed to assist in modeling and/or analysis recognize and leverage this view by enabling *compositional analysis* of networks of gadgets. This then facilitates the checking of safety properties or the inference of conditions or constraints under which safe operation is guaranteed.

Towards the stated goals, we propose in this paper a methodology and tool (NetSketch) for specifying and analyzing large network flow problems.

II. Compositional Analysis in NetSketch

As a tool, NetSketch supports *compositional* (in contrast to *whole-system*) analysis, which is additionally incremental (distributed in time) and modular (distributed in space). Schematically and somewhat simplistically, we can contrast *whole-system* and *compositional* analyses according to Figure 1, where "$[\![x]\!]$" denotes "the analysis of object x", "$\otimes$" an associative operation for connecting two components of a larger network, and "$\star$" an associative operation for combining two analyses.

Here it is important to note that for an analysis to be compositional, it must allow inter-checking of gadgets to happen in *any* order, thus enabling more flexible patterns of development and update. This stands in sharp contrast to modular analysis, which may prescribe a *particular* order in which the modules have to be analyzed. [2]

Analysis of Incomplete or Sketchy Specifications: By its nature, whole-system analysis cannot be undertaken if a gadget (such as B in Figure 1) is missing or if it breaks down (indicated by the question mark "?"). Moreover, if the missing gadget is to be replaced by a new one (B' in Figure 1), whole-system analysis must be delayed until the new gadget becomes available for examination and then the entire network must be re-analyzed from scratch. If we are interested in certifying that a particular invariant is preserved throughout the network without running into the limitations of whole-system analysis – specifically, inability to deal with incomplete or "sketchy" topologies and/or incurring the cost of having to re-examine the entire network – and if we can formalize this invariant using type-theoretic notions at the interfaces of gadgets (denoted by $\langle _ \rangle$ in Figure 1), then we can adopt the alternative approach of compositional analysis, which is *not* invalidated by the presence of *holes* (the empty interfaces $\langle _ \rangle$ in Figure 1). Simply put, one can think of a "hole" as a placeholder where a system integrator can place different gadgets satisfying the same interface types, interchangeably and at different times.

Our schematic comparison above, between compositional and whole-system analyses, calls for an important proviso if we are to reap the benefits of the former. The cost of combining two analyses (via the operation "$\star$" in Figure 1) should be significantly smaller – specifically, below a computational complexity that is acceptable to the user – than the cost of combining two networks (via the operation "$\otimes$" in Figure 1) and then analyzing the combination again from scratch. However, even with that proviso and the additional proviso that all the pieces (gadgets) of a network are in place so that a whole-system analysis is at all an option, it will not be that compositional analysis always wins over whole-system analysis. An analysis – any analysis – is of a few properties of interest and, as such, an abstraction of the actual network. An analysis determines conditions under which the network can operate safely (relative to appropriately defined safety criteria). Within the parameters and limits of the modeling abstraction, an *exact analysis* is one that determines *all* conditions of safe operation. An exact analysis typically requires whole-system analysis and, as such, may be very expensive. But will its cost always outweigh its benefits? It depends. Reverting to a compositional and computationally feasible analysis may force additional abstraction, at the price of perhaps excessive and unacceptable approximation in the results, as we shall illustrate later. An *approximate analysis* will typically determine a proper subset of the conditions of safe operation and, as such, will be sound but not complete. The tradeoff offered to users will be between completeness and precision of results (typically via exact and whole-system analysis) and computational feasibility (typically via approximate and compositional analysis).

III. A Network Sketching DSL

Each gadget (*i.e.*, node) of a network flow may impose constraints on its respective inputs and outputs; a topology coupled with its entire constraint set form an exact model, and a whole-system analysis of the network must solve the constraint set for the given topology. Our compositional approach uses *types* to approximate (*i.e.*, sketch) the constraints on each gadget's interfaces. Our DSL is used to describe the connectivity of gadgets (and holes) and to infer and check the types across the network.

To illustrate and motivate the need for our DSL, consider a particular network flow application, namely vehicular-traffic networks, where types of interest are *velocity types* and *density types*. A simple version of such types can be formalized as non-empty intervals over the natural numbers, each denoting a range of permissible velocities or densities. Velocity and density types can be inferred in an inside-out fashion, starting from the constraints regulating traffic at each of the nodes in the network. Such constraints can be formalized as equalities and inequalities of polynomial expressions over velocity and density parameters.

Suppose $\mathcal{M}$ and $\mathcal{N}$ are traffic flow networks of some sort – here "traffic flow" may equally refer to the flow

[2] A good example of the difference between modular and compositional analysis is provided by type inference for **ML**-like functional languages. Type inference is one approach to analyze programs statically. **ML**-like type inference is modular but not compositional.

CONNECT

$$\frac{\Gamma \vdash \mathcal{M} : (\mathsf{In}_1, \mathsf{Out}_1) \qquad \Gamma \vdash \mathcal{N} : (\mathsf{In}_2, \mathsf{Out}_2)}{\Gamma \vdash \mathcal{M} \,\square\, \mathcal{N} : (\mathsf{In}_1, \mathsf{Out}_2)}$$

where if $\mathsf{Out}_1 = \langle \sigma_1, \ldots, \sigma_n \rangle$ and $\mathsf{In}_2 = \langle \tau_1, \ldots, \tau_n \rangle$ then $\sigma_1 <: \tau_1, \ \ldots \ , \sigma_n <: \tau_n$

LET

$$\frac{\forall\, k \in \{1 \ldots n\} :\ \Gamma \vdash \mathcal{M}_k : (\mathsf{In}_k, \mathsf{Out}_k) \qquad \Gamma, X : (\mathsf{In}', \mathsf{Out}') \vdash \mathcal{N} : (\mathsf{In}, \mathsf{Out})}{\Gamma \vdash \textbf{let}\ X \in \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}\ \textbf{in}\ \mathcal{N} \ : (\mathsf{In}, \mathsf{Out})}$$

where $\mathsf{In}_1 = \cdots = \mathsf{In}_n = \mathsf{In}'$ and $\mathsf{Out}_1 = \cdots = \mathsf{Out}_n = \mathsf{Out}'$

Figure 2. Examples of two rules from NetSketch DSL Specification.

of packets in a communication networks, the flow of data tupples in a stream database, or the flow of vehicles in a network of roads. Suppose $\mathcal{M}$ has the same number n of output (exiting) links as $\mathcal{N}$ has of input (entering) links, and both are given as ordered sequences of length n. Suppose $\mathcal{M} : (\mathsf{In}_1, \mathsf{Out}_1)$ and $\mathcal{N} : (\mathsf{In}_2, \mathsf{Out}_2)$ are typings of $\mathcal{M}$ and $\mathcal{N}$ assigning appropriately defined types to their input/output links. The formal syntax of our strongly-typed DSL will be defined by rules of the form shown in Figure 2.

The side condition of the rule CONNECT is written right after it, stating that to safely connect the output links of $\mathcal{M}$ to the input links of $\mathcal{N}$, the output types of $\mathcal{M}$ must be subtypes of the corresponding input types of $\mathcal{N}$. Figure 2 shows another rule for the LET construct, which formalizes the idea that, in a hole X of a network $\mathcal{N}$, we can place at will any of n different networks $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ as long as they satisfy the same interface types.

The above two rules are presented to illustrate the nature of NetSketch's underlying formalism. Refinements and generalizations of these two rules, as well as several others, which we employ "under-the-hood" from within the tool, constitute the formalism. Collectively, they define the formal syntax of NetSketch's network flow DSL. We refer interested readers to a companion paper [1] for a full definition of the NetSketch formalism.

With a DSL and constraints of the form just described, we can enforce various desirable properties across, for example, a vehicular-traffic network, such as *no backups* (traffic is not piling up at any of the links entering a node at any time), *fairness* (there is no link along which traffic is permanently prevented from moving, though it may be slowed down), *conservation of flow* (entering traffic flow in a network is equal to exiting traffic flow), *no gridlock* (mutually conflicting traffics along some of the links ultimately result in blocking traffics along all links), *etc.*

There is more than one reasonable way to formalize the semantics of network typings. We consider two, corresponding to what we call "weak validity" and "strong validity" of typed specifications. A gadget typing is considered *weakly valid* if for every traffic satisfying the input types there exists a way of channelling traffic through the gadget, consistent with its internal constraints, that satisfies the output types. *Strongly valid* types are such that *every* way of channeling traffic under the input types satisfies the output types.

From Modules and Gadgets to Network Sketches: In our formalism, a *module* corresponds to the basic building block of a flow network. Modules are fully specified in the sense that exact (tight) constraints characterizing their safe operation (*e.g.*, invariants relating parameters associated with their input and output links) are known *a priori*. The specific mechanism via which exact characterizations of modules are acquired is an orthogonal issue: They may be the outcome of a whole-system analysis; they may be distilled from implementation code; they may be lifted from data sheets; or they may be simply assumed. NetSketch *gadgets* are inductively defined: A module is a (base) gadget, a hole is a gadget, and any interconnection of gadgets is itself a (network) gadget. For ease of exposition, we use *network* to refer to a network gadget.

The definition of a network implies that (unlike modules), networks admit incomplete specification by allowing for holes. More importantly, networks may be typed in the sense that specific constraints or invariants at their interfaces do not have to be exact – *i.e.*, such type constraints may allow for looser bounds than what is absolutely necessary. As such networks can be seen as approximations of the systems they model, and it is in that sense that they constitute "sketches" of the system being modeled or analyzed. Such approximations may arise as the result of trading off whole-system for compositional analyses, and/or trading off exactness of analysis for computational efficiency and scalability. As we alluded before, exposing this tradeoff is one of the main design goals of NetSketch.

IV. THE NETSKETCH TOOL

NetSketch presents its user with two modes of operation: *Base* and *Sketch*. These reflect the granularity of the description within the tool (*Unyped* and *Typed*) and whether whole-system analysis or compositional analysis will be employed, respectively. In the base mode of operation, NetSketch's interface allows users to describe exact (typically small) specifications of gadgets consisting of connected components for which whole-system analysis is viable. In the sketch mode of operation, NetSketch allows users to describe and explore network gadgets for which compositional analysis is desired. The NetSketch operational process is illustrated in Figure 3.

In the base mode of operation, a user defines a graph topology by selecting from predefined *classes* of network gadgets (of which she may define her own) and by graphically drawing connections between these gadgets. The topology of these gadgets and their respective edges form a graph of constraints. Prior to entering the sketch mode of operation, NetSketch performs an analysis of the gadget constraint set and presents to the user a simplified ("black box") representation of the gadget graph. The sketch interface for this representation provides the user with scalar bounds as input and output types derived from the constraint set with respect to some specific target criteria. Once in sketch mode, the user may further refine or constrain the current network sketch (*i.e.*, return to the base (untyped) mode to consider other constraint criteria), or investigate the connection of other existing networks to the current network, including

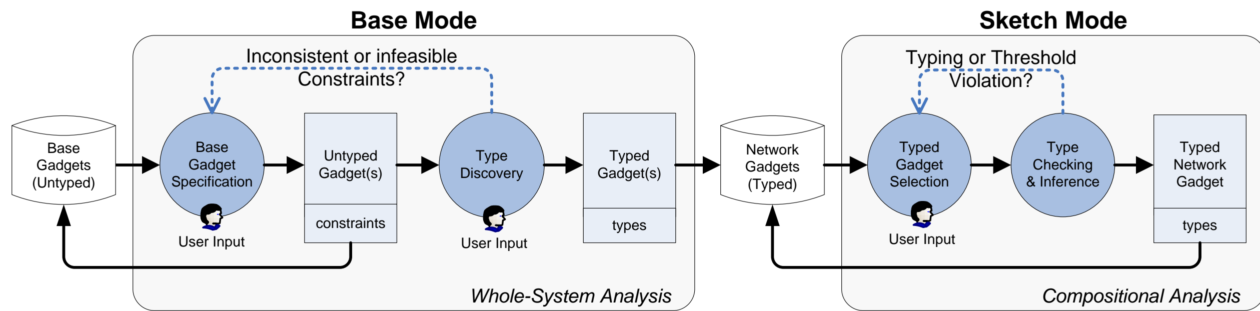

Figure 3. A process overview of Network Flow modeling and assessment with NetSketch.

the specification and analysis of "holes" (placeholders for future gadgets) in the topology.

Manipulating Base (Untyped) Gadgets: The specification (modeling) of base gadgets in NetSketch is done graphically by placing and connecting instances of gadget classes. Each gadget class defines the number of ingress and egress ports for all gadgets of this class, as well as the generic, relational constraints between the inputs and outputs. For example, in a vehicular traffic domain, a gadget class "Merge" may take two inputs, produce one output and have the generic constraint that the vehicular output density (*i.e.*, number of vehicles) is equal to the sum of the inputs ($out0 = in0 + in1$). A user may define a base gadget (or a class of gadgets) on-the-fly within the tool and this gadget definition can be used immediately or saved for future reuse. The specific gadgets that are available for placement on the canvas consist of those previously defined by users (or other domain experts). Users may ultimately build a library of gadget definitions that form a domain-specific operating context.

The placement of a gadget class on the canvas creates a gadget instance. The ports of a gadget instance are populated with new constraint variables and generic constraints are instantiated using these specific variables and added to the constraint set. When a user inserts a gadget instance, the system immediately prompts for specific numeric bounds on these variables (if possible) to distinguish a specific instance of a gadget from the generic class of gadget. Gadget instances may be connected to other instances on the canvas via edge placement. Edge placement imposes equality constraints on the ports at the ends of the edge. To improve readability, all references to the head variable in the constraint set (and UI) are replaced with the tail variable.

As a user places gadgets and/or edges, she will see the direct effects of these changes in the constraint set presented in the bottom frame of the GUI. The constraint set is stored with respect to the gadget that introduced the constraints. If a gadget is removed from the topology, the corresponding constraints may be removed also. When an edge is deleted the variables at the head and tail must be re-separated and some gadget constraints may be updated as a result.

Beyond stand-alone, direct definition and specification of gadget classes, classes may also be created by folding connected gadget instances together to create a reusable *envelope*: a visual simplification of a gadget topology. An envelope is rendered as a single node that contains all of the constraints of its constituent gadget and edges. An envelope exposes all non-connected ports (both to and from the collection of gadgets) as inputs and outputs of the envelope. The constraints for the gadget within the envelope (and their edges) become the generic constraint set for the envelope. In order to make the envelope sufficiently abstract, some of the constraints that are automatically folded into the envelope definition may be removed manually since specific bounds for some gadget instances may not apply to the entirety of the envelope class. Envelopes are useful both for abstracting particularly complex configurations of gadgets (*i.e.*, complex or unwieldy topologies) and for the promotion of reuse for commonly needed gadget instances. For example, a user may create a 3-way merge by connecting two merge gadgets and exporting an envelope of that model.

Manipulating (Typed) Network Gadgets: Using the sketch mode of operation involves placing and connecting *typed* gadgets (and holes) into a typed network. This enables a user to construct networks and to explore their connectivity and interface properties. The presentation and use of simplified type-centric (interval) interface lets a user immediately see the range of valid inputs into a network of typed gadgets, yet still safely ignore the complete set of constraints when connecting to other typed gadgets or networks.

To use an untyped gadget in the sketch mode, the Net-Sketch tool performs an automated whole-system analysis on the gadgets to produce types. While in the analysis phase, the constraint solver processes the current constraint set and assigns feasible bounds (if they exist) to constituent gadgets. If the user specifies optimization criteria/constraints (*e.g.*, maximize a specific edge, verify that flow is conserved) when converting an untyped gadget to a typed one, the solver will attempt to produce a feasible range with respect to such constraints. If no constraints are specified, the analysis tool tries to find the widest bounds (types) for the given untyped gadget. Should the whole system analysis fail to find a feasible solution to provide the basis for types, the user must either adjust the gadgets or the target criteria.

Within the sketch mode of operation, the user may load other typed gadgets or networks from the repository (or load and convert other untyped gadgets into gadgets) and the system can suggest possible placement locations (and restrict illegal placements) for new edges based on the visible (and saved) type information of these gadgets. This behavior illustrates the benefit of a type-centric interface: depending on the shape of the constraint set within the untyped gadgets, it may not have been possible to perform the same analysis when connecting modules directly.

Transitioning from Base to Sketch Mode: An important "decision point" for a NetSketch user is choosing the point at which to abandon whole-system (exact) analysis and to switch to compositional (sketched) analysis (*i.e.*, transition from base to sketch modes of operation). Similarly, when converting an untyped gadget of any size greater than one to a typed interface, the user must choose the gadget granularity in the resulting network (whether to make each untyped gadget an individual typed gadget, to combine all gadgets into a single typed gadget, or to group untyped gadgets into typed gadgets). For instance, while a user may decide to transition each individual untyped gadget into a stand alone typed gadget for, say, a preference for the simpler interface, such a choice will result in a loss of specificity, which may not be warranted. Thus, the time of transition from base to sketch modes, and the choice of typed gadget granularity should be considered carefully.

Independent of user choices, at some point the constraint sets in a topology of untyped gadgets may become sufficiently complex that compositional analysis becomes the *preferred* (if not only) possibility for analysis. We identify this point using the constraint *threshold*, which may be set in a number of ways – *e.g.*, based on number of gadgets, number of edges, number of constraints, number of variables within the constraints, time taken to bound the feasible region of the solution, the shape of the constraints. Automatically determining this threshold given the shape/complexity of a constraint set (and the use of various non-linear programming libraries) is planned future work.

Manipulating Holes: A hole is a placeholder for any unknown or under-specified gadget or network. Holes enable modeling and verification to proceed even if only part of the system is known. While the specification of a topology with incomplete information may seem contrived, many valuable usage scenarios for NetSketch (and underlying formalisms) stem from the ability to assess safe component replacement within existing topologies and the ability to determine the interface properties of an unbound gadget or network so that other valid gadgets (or combinations thereof) may substitute it. NetSketch's compositional analysis and verification engine may be used to infer type constraints for any such holes from the typed network and, in so doing, essentially indicates which typed gadgets or networks may be later used in the location of the hole. This usage of holes is inspired from (and directly analogous to) the inference of types for variables in programming languages. While NetSketch does not automatically find and suggest all possible gadgets that fit in a given type signature, the tool *will ultimately* allow the user to attempt to place a network into a hole, and will use type information to permit or restrict such placement.

Implementation Details: NetSketch has been implemented in Java 1.6 and uses the JGraphX (JGraph 6) Open Source graph drawing component [2] libraries to facilitate graph visualization. The GNU Linear Programming Toolkit (GLPK) [3] is used to solve the constraint sets that are built as a result of the composition of modules in the GUI.

V. Use Case 1: Task Scheduling

The generality of the NetSketch formalism is such that it can be applied to problems that are not immediately apparent as constrained-flow network problems. For illustrative purposes, consider a single processor scheduled via EDF. Periodic tasks each require c_i time units of computation within their respective fixed periods of t_i time units. A simple notion of safety here is the schedulability test that the sum of all utilizations (c_i/t_i) is less than or equal to 100%. One way to model this domain in NetSketch requires two gadgets classes. The first gadget class required would be used to represent the individual tasks to be scheduled; this gadget class would have two inputs (one for c_i and one for t_i) and would produce a single output representing c_i/t_i. The second necessary gadget class accepts arbitrarily many inputs and produces a single output. This gadget would have a constraint that the output is equal to the sum of the inputs, and another that reflects that the output is within the range [0,100].[3]

Even with these simple constructs, one may consider several interesting usage scenarios: (1) Swapping the schedulability test gadget to that of another scheduling policy –*e.g.*, a Rate Monotonic Scheduling policy, which would have a constraint that the combined utilizations is $<= n(2^{(1/n)}-1)$; (2) Investigating the remaining utilization of a task set by generating types against a specific task set and then placing a hole as an input to the test gadget; (3) Allowing the "supply" of cycles, (*i.e.*, total available utilization) to also be an input of the gadget to model (*e.g.*, a virtual server that is able to produce x time units every y time units), requiring more involved constraints. More complicated gadgets can be constructed for richer task models (*e.g.*, allowing for a maximum number of deadlines over a window), virtualized resources (*e.g.*, periodic servers), as well as more elaborate schedulers (*e.g.*, statistical RMS, pinwheel scheduling).

VI. Use Case 2: Vehicular Traffic

An engineer working for a large metropolitan traffic authority has the following problem. Her city lies on a river bank across from the suburbs, and every morning hundreds of thousands of motorists drive across only a few bridges to get to work in the city center. Each bridge has a fixed number of lanes, but they are all reversible: the operator has the ability to decide how many lanes are available to inbound and outbound traffic during different times of the day. The engineer must decide how many inbound lanes to open in the morning with the primary goal of ensuring that no vehicular backups occur in the city center, with a secondary goal of maximizing the amount of traffic that can get into the city.

The city street grid is a network of a large number of only a few distinct kinds of traffic junctions: fork, merge, and crossing junctions. We call streets with traffic going into and out of a junction *links*. Both the structure of each kind of junction and the problem the engineer must solve can be modeled using (untyped) base gadgets. Once she has specified the entire network, she may switch to a (typed) network gadget. The types assigned to each gadget and the links into and out of the typed network gadget can help her decide how many lanes to open on each bridge while ensuring no backups.

[3]Scaling this range by 100 avoids non-integer values.

Example 1: A *fork* has one incoming link (call it 1) and two outgoing links (call them 2 and 3). This traffic junction can be modeled using a gadget (call it $\mathcal{A}_{\mathbf{F}}$) consisting of a single node (call it **F**). Let link i be associated with velocity parameter v_i and density parameter d_i, for $i \in \{1, 2, 3\}$, and Con the set of constraints associated with $\mathcal{A}_{\mathbf{F}}$. We partition Con into two subsets, $\mathsf{Con}_{\text{nodes}}$ (constraints regulating traffic through **F**) and $\mathsf{Con}_{\text{links}}$ (lower and upper bounds on $v_1, d_1, v_2, d_2, v_3, d_3$). The set $\mathsf{Con}_{\text{nodes}}$ contains predefined constraints that are required for all *fork* gadgets:

$$(1)\ d_1 = d_2 + d_3; \quad (2)\ d_1 * v_1 \leqslant d_2 * v_2 + d_3 * v_3$$

Constraint (1) enforces *conservation of density* when **F** is neither a "sink" nor a "source", whereas constraint (2) encodes the *non-decreasing flow* invariant, namely that traffic along exit links may accelerate. Notice that constraint (1) is *linear* constraint, while (2) is *quadratic*. Notice also that these constraints are mutually consistent, *i.e.*, simultaneously satisfiable by a particular valuation (an assignment of values to the 6 parameters $d_1, v_1, d_2, v_2, d_3, v_3$). Combined, these constraints ensure that traffic is not piling up at the fork entrance. Strictly speaking, we should also add a constraint of the form: "If $v_2 * d_2 + v_3 * d_3 > 0$ then $v_1 * d_1 > 0$" or, given that $d_1 = d_2 + d_3$, "If $v_2 + v_3 > 0$ then $v_1 > 0$", *i.e.*, if exiting traffic flow $\neq 0$ then entering traffic flow $\neq 0$. Our syntax of constraints does not allow the writing of conditional constraints of this form. However, if we assume $v_1 \neq 0$, a reasonable assumption, this conditional constraint is already implied by constraints (1) and (2). The set $\mathsf{Con}_{\text{links}}$ specifies lower and upper bounds on the parameters, *i.e.*, it consists of constraints of the form $a_p^{\text{lo}} \leqslant v_p \leqslant a_p^{\text{up}}$ and $b_p^{\text{lo}} \leqslant d_p \leqslant b_p^{\text{up}}$ for $p \in \{1, 2, 3\}$, where $a_p^{\text{lo}}, a_p^{\text{up}}, b_p^{\text{lo}}, b_p^{\text{up}}$ are particular scalar values.

Other meaningful constraints may be introduced into $\mathsf{Con}_{\text{nodes}}$ to alter the goal of the construction:

– Balanced densities at exits: $d_2 \leqslant d_3 \leqslant d_2 + 1$
– Balanced flows at exits: $d_2 * v_2 \leqslant d_3 * v_3 \leqslant d_2 * v_2 + 1$
– Constant velocities from entry to exit: $v_1 = v_2 = v_3$
– Conservation of kinetic energy: $d_1 * v_1^2 = d_2 * v_2^2 + d_3 * v_3^2$

Before we specify the other two junction types, suppose that the junction connected directly to a bridge is a fork junction. A typed specification for the fork gadget would consist of an assignment of types to the input and output links. If the problem were trivial, containing only one bridge and one junction, this type would specify exactly the bounds that would guarantee no backups and could be used to decide exactly how many lanes to open on the bridge.

Example 2: A *merge* junction has two incoming links (call them 1 and 2) and one outgoing link (call it 3). The corresponding gadget (call it $\mathcal{A}_{\mathbf{M}}$) for this junction is very similar to the fork module in Example 1. There is a single node, call it **M**. We omit the details and justifications for this gadget, which are similar to those in Example 1.

Example 3: A *crossing* junction has two incoming links (call them 1 and 2) and two outgoing links (call them 3 and 4). The corresponding gadget (call it $\mathcal{A}_{\mathbf{X}}$) again has a single node, call it **X**. Let link i be associated with velocity parameter v_i and density parameter d_i, for $i \in \{1, 2, 3, 4\}$, and Con the set of constraints associated with $\mathcal{A}_{\mathbf{X}}$. Again here, we partition Con into two subsets, $\mathsf{Con}_{\text{nodes}}$ (constraints regulating traffic through **X**) and $\mathsf{Con}_{\text{links}}$ (lower and upper bounds on $v_1, d_1, v_2, d_2, v_3, d_3, v_4, d_4$). The constraints in $\mathsf{Con}_{\text{nodes}}$ can be of different kinds, depending on different considerations, such as whether or not the incoming traffics, through links 1 and 2, are given a choice to exit through link 3 or link 4. We restrict attention in this example to the simple case when there is no such choice: All traffic entering through link 1 must exit through link 3 and at the same velocity, and all traffic entering through link 2 must exit through link 4 and at the same velocity. This is expressed by four constraints:

$$v_1 = v_3; \quad v_2 = v_4; \quad d_1 = d_3; \quad d_2 = d_4$$

If the total density of entering traffic, namely $d_1 + d_2$, exceeds a "jam density" that makes the two entering traffics block each other, there will be backups. We therefore presume there is an upper bound, say 10, on $d_1 + d_2$ below which the two traffics do not impede each other and there are no backups as a result:

$$d_1 + d_2 \leqslant 10$$

Below a total density of 10, we can imagine that the two incoming traffics are sparse enough so that they smoothly alternate taking turns to pass through the crossing junction.

The modeling of a crossing in this example makes all the constraints in Con *linear*. More complicated situations, enforcing additional desirable properties besides no-backups, will typically introduce non-linear constraints such as those listed in Example 1.

The entire city grid can be modelled by a network $\mathcal{N}$ of connected instances of the typed gadget $\mathcal{A}_{\mathbf{F}}$, $\mathcal{A}_{\mathbf{M}}$, and $\mathcal{A}_{\mathbf{X}}$. The incoming edges of $\mathcal{N}$ would represent the bridges. Since the constraints that restrict the intervals for the link parameters in each individual gadget instance guarantee no backups (thanks to the fact that the inference rules are sound), this guarantee also holds for links of the composed network $\mathcal{N}$ that consists of these gadget instances. Thus, as soon as the best (widest) types are inferred for the incoming links to $\mathcal{N}$, the engineer can set the number of lanes of traffic in a manner that respects these bounds and she can be certain that no backups will occur in the city center.

It is possible that the types generated in this process will not allow any traffic to flow into the city. In this situation, the engineer always has the option of loosening the constraints specified for each module, and trying again.

Example 4: This example is more complicated than the preceding ones in this section. Consider our engineer modeling a traffic module that represents the center of the city, which is accessible by three inbound lanes and three outbound lanes. The untyped gadgets that she connects now (that will ultimately be the network $\mathcal{M}$) are based on the previous three example gadgets. $\mathcal{M}$, consists of three gadgets named A, B, C and has 10 links named $1, 2, \ldots, 10$.

To complete the (untyped) specification of $\mathcal{M}$, we associate parameters with each of its links and define a set Con (constraints over these parameters). For simplicity in this example, we restrict attention to density parameters, ignoring velocity parameters. The assignment of density parameters to the links of $\mathcal{M}$ is shown in Figure 4.

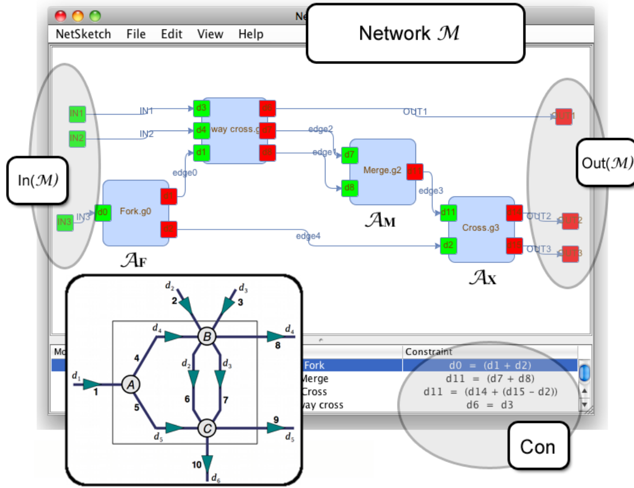

Figure 4. The traffic module from Example 4, shown abstractly (inset) and from within NetSketch (main).

As in preceding examples, the constraints in Con specify relationships between incoming and outgoing densities at each junction, as well as lower and upper bounds on these densities. The NetSketch GUI being used to define $\mathcal{M}$ is shown in Figure 4. The instance of B shown is a user-defined gadget, while C which is shown as a single node in Figure 4 (inset) has been modeled in the tool using two gadgets: a Merge junction and a Crossing junction. While assembling the untyped gadgets, our user has the following constraint set automatically constructed (as defined by the gadget class):

$$d_1 = d_4 + d_5 \quad (A); \qquad d_2 + d_3 + d_4 \leqslant 10 \quad (B)$$
$$d_2 + d_3 = d_6 \quad (C); \qquad d_2 + d_3 + d_5 \leqslant 10 \quad (C)$$

Only the lower and upper bounds on the density parameters must be specified by our engineer, and she is prompted to provide them after placing each module on the canvas.

$$2 \leqslant d_1, d_4, d_5, d_6 \leqslant 8 \qquad 0 \leqslant d_2, d_3 \leqslant 6$$

If our user decides to convert this untyped gadget topology to a Typed Network to analyze the density bounds for traffic into and out of the city center, the user supplied bounds alone are insufficient. The user supplied bounds do not constitute a *valid* type for the inputs and outputs of a Typed Network.[4] Instead, the tool prompts our user for an objective function to try to find a specific valid typing for each of the constituent gadgets. Assuming the engineer's city center is connected to a large thoroughfare via d_6, she may generate types that abide by this request. Moreover, should she have models of the rest of the city, but not the portion of roadway that connects to d_4, she may use a hole connected to d_4 in her Network. The tool will indicate the range of densities expected to enter that hole in the event that d_6 is maximized.

VII. Use Case 3: Video Streaming

As another use case of NetSketch, we consider the problem of video stream aggregation into a constant bit rate pipe, *e.g.*, into the upstream bandwidth of a video server. To safely serve a set of video streams, we must ensure that in any period of time the video streams do not exhibit an aggregate rate that is larger than that of the pipe. This problem would be quite simple if the bit rates of the video streams were fixed. This is not the case for current video encoding standards, which exhibit highly variable bit rates due to the different types of underlying frames (*e.g.*, MPEG-2/4 I, P, and B frames).

Let $a(t)$ denote the cumulative number of bits for a video stream. Two important parameters describing a video stream are the mean bitrate and the peak bitrate:

$$\text{Mean Rate} = \frac{a(t_f)}{t_f}; \quad \text{Peak Rate} = {}^{max}_{\;\;i} \left\{ \frac{a(t_{i+1}) - a(t_i)}{t_{i+1} - t_i} \right\}$$

where t_i is the time of the ith frame and t_f is the time of the last frame (assuming that the stream starts a $t_1 = 0$).

Using the stream's mean rate for reserving resources is not practical as it does not provide a small enough bound on the playback delay and, potentially, may require a very large buffer to avoid buffer underruns at the receiver. On the other hand, while using the peak rate would give the minimum playback delay and would minimize the amount of buffering required, it also wastes resources because bandwidth utilization will be very low, making it impossible to scale the system to a large number of streams. To deal with this dilemma, one may use the *effective bandwidth* as a way to characterize (using a tight constant rate envelope) any time interval of the stream, so that the buffering delay experienced during this interval is bounded. One way to characterize the effective bandwidth is by specifying a rate r for the stream as well as the maximum burst size s that is possible under that rate, as well as the minimum window of time w necessary for such a burst to build up (which is typically well defined for encoding standards – *e.g.*, a GoP for MPEG-2/4).

For a given stream, multiple (r, s, w) values may exist, reflecting a tradeoff between bandwidth and delay. In particular, if r is the peak rate (as described above) then there will be no need for any buffering at the server, since there will always be reserved uplink capacity to immediately serve the content, *i.e.*, $s = 0$ and $w = 1$. Similarly, if r is the mean rate (or less) then the corresponding values of s and w will increase. In general one can see that: $r + s/w < r_{max}$ where r_{max} is the peak rate mentioned above.

Aggregation Gadget: While in practice one would be interested in the aggregation of a set of video streams, the basic building block we consider is that of aggregating two video streams (or video stream aggregates) on a server. Such a gadget would have two incoming links ($\mathsf{In} = \langle 1, 2 \rangle$) and one outgoing link ($\mathsf{Out} = \langle 3 \rangle$). The incoming links capture the properties of the two streams to be aggregated, whereas the outgoing link captures the properties of the aggregated stream. We note that the aggregation gadget induces a relationship between the three links that could be specified using the following relationships:

$$r_3 = r_1 + r_2; \; s_3 = s_1 + s_2; \; w_3 = min(w_1, w_2)$$

Smoothing Gadget: Another operation that one may perform on a video stream (or an aggregate thereof) at the server

[4] For a more complete treatment invalid, weakly valid, strongly valid, and optimal types, we refer the reader to our companion paper [1].

is smoothing. Smoothing a video stream is done through buffering (and hence introducing an end-to-end delay). Thus, a smoothing gadget with a buffer of size b would have one incoming link ($\mathsf{In} = \langle 1 \rangle$) and one outgoing link ($\mathsf{Out} = \langle 2 \rangle$) subject to the following constraint relating the characteristics of the incoming and outgoing links:

$$r_2 = r_1;\ s_2 = max(s_1 - b, 0);\ w_2 \leq w_1$$

Transmission Gadget: Finally, in order to transmit a video stream (or an aggregate of video streams) from the server with a constant-bit-rate uplink capacity c, we define a gadget with one incoming link ($\mathsf{In} = \langle 1 \rangle$) and one outgoing link ($\mathsf{Out} = \langle 2 \rangle$) subject to the following constraint relating the characteristics of the incoming and outgoing links:

$$r_2 = r_1;\ s_2 = s_1;\ w_2 = w_1;\ r_1 + \frac{s_1}{w_1} \leq c$$

Example 5: Now, consider a user with an existing video streaming network who wants to find out the "maximal" stream that can be "inserted" without violating any existing constraints of on-going streams. In its simplest form, this could be an aggregation gadget where the output link as well as one of the input links are "specified" (constrained), leaving the second input link as a hole.

Example 6: As a second example, consider an aggregated set of streams which need to be transmitted but, to match the aggregate to the transmission link, the aggregate must be smoothed through buffering. Notice that smoothing reduces the size of the (aggregate) bursts, which in turn reduces the requirement on the capacity of the transmission gadget. In this topology a user designates a hole (*i.e.*, an unspecified smoothing gadget) ahead of the transmission gadget and, given a particular capacity, we can find the minimal smoothing necessary for the plumbing to work.

The constraints associated with the gadgets presented above only exemplify safe constraints that could be asserted by a programmer or system integrator. These constraints could be refined and/or made "tighter" (*i.e.*, more permissive). By introducing additional "variables" (*e.g.*, delay or loss rates), the gadgets could also be made more faithful to specific implementation details, *e.g.*, using concepts from Network Calculus to establish relationships between flows.

VIII. Related and Future Work

Previously proposed systems for reasoning about the behavior of distributed programs ([4], [5], [6], [7], and [8], [9]) retain extensive details about the *internals* of a system's components in assessing their interaction. While this improves expressive power, analyses become inherently unmodular. Details are not easily added to or shed from a model when it is interfaced with another. In a global analysis, the specification of components is highly coordinated and inevitably wedded to particular methodologies. Not being sufficiently *general*, such specifications preclude analysis of interactions between components specified using different methodologies. Furthermore, it is not generally possible to analyze parts of a system independently and then, ignoring their internals, assess if those parts can be assembled.

An essential functionality of NetSketch is the ability to reason about, and find solution ranges that respect, sets of constraints. In its general form, this is the widely studied *constraint satisfaction problem* [10]. NetSketch types are linear constraints, and linear constraint satisfaction is a classic problem for which many documented algorithms exist. A distinguishing feature of NetSketch is that it does not treat the set of constraints as monolithic. Instead, a tradeoff is made in favor of providing users a way to manage large constraint sets through abstraction, encapsulation, and composition. Complex constraint sets can be hidden behind simpler constraints (NetSketch types) in exchange for a more restrictive solution range. The conjunction of large constraint sets is made more tractable using compositional techniques.

This work extends and generalizes our work in TRAFFIC [11]), and complements our earlier work in CHAIN [12]).

NetSketch's current constraint system for untyped gadgets (limited to linear constraints) is a proof-of-concept enabling our work on typed networks (holes, types and bounds). We are working to extend the constraint solver to support more complex untyped gadgets. There is also a need to automatically assess systems of non-linear constraints and determine the *threshold*. As indicated in the accompanying paper on the NetSketch formalism [1], there is no natural ordering of types for sketches. The algorithm for assigning types can be amended to consider a measure and ordering on types reflecting application-dependent objective functions.

References

[1] A. Bestavros, A. Kfoury, A Lapets, and M. Ocean, "Safe Compositional Network Sketches: Formalism," CS Dept., Boston University, Tech. Rep. BUCS-TR-2009-029, September 29 2009. [Online]. Available: http://www.cs.bu.edu/techreports/2009-029-netsketch-formalism.ps.Z

[2] JGraph Ltd., "JGraph: The Java Open Source Graph Drawing Component," http://www.jgraph.com/jgraph.html.

[3] Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia, "GNU Linear Programming Kit," http://www.gnu.org/software/glpk/.

[4] J. Baeten and W. Weijland, *Process Algebra*. Cambridge University Press, 1990.

[5] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes (Part I and II)," *Information and Computation*, no. 100, pp. 1–77, 1992.

[6] N. Lynch and M. Tuttle, "An introduction to input/output automata," *CWI-Quarterly*, vol. 2(3), no. 3, pp. 219–246, Sep. 1989.

[7] N. Lynch and F. Vaandrager, "Forward and backward simulations – part I: Untimed systems," *Information and Computation*, vol. 121(2), pp. 214–233, Sep. 1995.

[8] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 1–17, May 1997.

[9] G. J. Holzmann and M. H. Smith, "A practical method for verifying event-driven software," in *Proc. ICSE99*, Los Angeles, CA, May 1999, pp. 597–607.

[10] E. P. K. Tsang, *Foundations of Constraint Satisfaction*. London and San Diego: Academic Press, 1993.

[11] A. Bestavros, A. Bradley, A. Kfoury, and I. Matta, "Typed Abstraction of Complex Network Compositions," in *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP'05)*, Boston, MA, November 2005. [Online]. Available: http://www.cs.bu.edu/faculty/matta/Papers/icnp05.pdf

[12] A. Bradley, A. Bestavros, and A. Kfoury, "Systematic Verification of Safety Properties of Arbitrary Network Protocol Compositions Using CHAIN," in *Proceedings of ICNP'03: The 11th IEEE International Conference on Network Protocols*, Atlanta, GA, November 2003.