# 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 so as to retain sufficient enough details 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 approach based on a strongly-typed, Domain-Specific Language (DSL) to specify network configurations at various levels of sketchiness along with invariants that need to be enforced thereupon. In this paper, we overview NetSketch, highlight its salient features, and illustrate how it could be used in applications, including 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.**

## 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 in a wholistic manner. For example, the development of applications with predictable timing properties necessitated the use 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, specific scheduling parameters, among many others. While justifiable in some instances, this bottom-up, vertical approach to establishing trust 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 they rely on. This horizontal approach to system design and development 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, in that context, we recognize three specific challenges that the work we present in this paper aims to mitigate.

**Exposing Tradeoffs:** The environments and tools supporting large-scale system integrators must expose the inherent tradeoff between the *exactness* of safety analysis with respect to the *specifity* of the underlying subsystems, and the *computational complexity* necessary for automated analysis. For example, it should be possible for a system integrator to under-specify, or *sketch* whatever guarantees or constraints are expected to hold in a subsystem, 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 for system-wide safety properties to hold. Similarily, it should be possible for a system integrator to escalate the automated analysis of safety properties based on the computational cost of such an analysis, perhaps opting for *sketchier* but cheaper analysis for less critical functionalities (or early on in the design phase).

**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 deep knowledge of) particular domain expertise. As we alluded earlier, while acceptable and perhaps expected for vertically-designed, 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, but also it must 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, less accessible domain-specific expertise (*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). 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 otherwise a regulator of flows that are characterized by 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). In a way, 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 to allow for checking of safety properties or for the inference of conditions or constraints under which safe operation can be guaranteed.

Towards the above-mentioned goals, in this paper we propose a methodology for the specification and analysis of large network flow systems. In section II, we highlight the prominent features of this methodology and of the formalism upon which it is based. Next, in Section III, we present a design (modeling and analysis) tool, called NetSketch, which we have developed in support of this methodology. In Sections V and VI, we present two illustrative use cases of the tool for shaping vehicular traffic networks and streaming video networks, respectively. We conclude the paper with a review of the related literature in Section VII and with a summary of current and future work in Section VIII.

## II. The NetSketch Framework and Formalism

In this section, we overview the salient features and the formal underpinnings of NetSketch. A significantly more detailed treatment of the formalism underlying NetSkecth (including proof of its soundness) is presented in a companion paper [1].

**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 constrast *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.[1]

**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 double question marks "??"). 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

[1]A good example of the difference between modular and compositional analysis is provided by type inference for **ML**-like functional languages. Type inference is a particular way of analyzing programs statically, one of several closely related approaches available today. **ML**-like type inference is modular but not compositional.

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

Figure 1. Contrasting whole-system and compositional analyses.

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 be operated 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 susbet of the conditions of safe operation and, as such, will be sound but not complete. The tradeoff offered to users will be between completeness or precision of results (typically via exact and whole-system analysis) and computational feasibility (typically via approximate and compositional analysis).

**A Domain Specific Language (DSL) for Sketching Network Flow Problems:** Each junction (or node) of a network may impose constraints on its respective inputs and outputs; the topology coupled with its entire constraint set form an exact model. 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.*, are sketched from) the constraints on each node's interfaces. Our DSL is used to describe the connectivity of nodes (and holes) and to infer and check the types.

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 a range of permissible 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 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$^{\square}$, as that of every rule, is written right after it. To safely connect the output links of $\mathcal{M}$ to the input links of $\mathcal{N}$, this side condition requires that the output types of $\mathcal{M}$ are 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 underlying formalism. Refinements and generalizations of these two rules, as well as several other rules, which we employ "under-the-hood" from within the tool, constitute the formalism that underly NetSketch. Collectively, they define the formal syntax of NetSketch's network flow DSL.[2]

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 of formalizing the semantics of network typings. We consider two, corresponding to what we call "weak validity" and "strong validity" of typed specifications. Let $\mathcal{M} : (\mathsf{In}, \mathsf{Out})$ be a typed specification for network $\mathcal{M}$, where $\mathsf{In}$ is an assignment of types to the input links and $\mathsf{Out}$ an assignment of types to the output links:

- $\mathcal{M} : (\mathsf{In}, \mathsf{Out})$ is *weakly valid* just in case, for every traffic entering $\mathcal{M}$, if the traffic satisfies the input types $\mathsf{In}$, then *there is* a way of channelling traffic flow through $\mathcal{M}$, consistent with its internal constraints, which will satisfy the output types $\mathsf{Out}$.
- $\mathcal{M} : (\mathsf{In}, \mathsf{Out})$ is *strongly valid* just in case, for every traffic entering $\mathcal{M}$, if the traffic satisfies the input types $\mathsf{In}$, then *every* way of channelling traffic flow through $\mathcal{M}$, consistent with its internal constraints, will satisfy the output types $\mathsf{Out}$.

Both kinds of validity are meaningful. The first presumes that nodes in the network communicate and cooperate, or that there is a network administrator with global knowledge, to optimally direct traffic through the network. The second presumes instead that nodes in the network are autonomous systems with restricted communication between them or communication limited to their immediate neighbors. These definitions are made precise and also more general, in [1], including notions of soundness and its related proof.

**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*.[3] 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.[4]

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 these tradeoff is one of the main design goals of NetSketch.

## III. The NetSketch Tool

### A. Overview of NetSketch Operation

NetSketch presents its user with two modes of operation: *Base* and *Sketch*. These modes reflect the granularity of

[2]Interested readers are referred to a companion paper [1] for a full specification of the NetSketch formalism.

[3]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 using domain-specific theories or calculi; they may be distilled from implementation artifacts; they may be lifted from data sheets; or they may be simply assumed.

[4]While the formalism underlying NetSketch has a clear need to distinguish between modules and networks, in our presentation of the NetSketch GUI, we use gadgets to refer to both, using the modifiers "Untyped" and "Typed" to make clear whether we are referring to an exact specification of an underlying subsystem, or to an approximate sketch thereof.

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

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

$$\textsc{Let}'\qquad \frac{\Gamma \vdash \mathcal{M}_1 : (\mathsf{In}_1, \mathsf{Out}_1) \ \cdots\ \Gamma \vdash \mathcal{M}_n : (\mathsf{In}_n, \mathsf{Out}_n) \qquad \Gamma, X : (\mathsf{In}', \mathsf{Out}') \vdash \mathcal{N} : (\mathsf{In}, \mathsf{Out})}{\Gamma \vdash \textbf{let } X \in \{\mathcal{M}_1, \ldots, \mathcal{M}_n\} \textbf{ in } \mathcal{N} : (\mathsf{In}, \mathsf{Out})}$$

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

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

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 (typically small) exact 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.

In the base mode of operation, a user defines a graph topology by selecting from predefined *classes* of network gadgets (of which he or she may define his or 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, presenting the user with a simplified (collapsed "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 the sketch mode of operation, the user may further refine or constrain the current network sketch (*i.e.*, return to the base (untyped) mode to consider other constraint criteria), investigate the connection of other existing networks to the current network, including the specification and analysis of "holes" (placeholders for future gadgets) in the topology,.

An overview of the NetSketch operational process is shown in Figure 3.

## B. Manipulating Base (Untyped) Gadgets

The specification (modeling) of base gadgets in NetSketch is done graphically through the placement and connection of 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 specific experts); users may ultimately build a library of domain related 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 asses to the constraint set. When a user inserts a gadget instance, the system immediately prompts a user to provide any specific numeric bounds on these variables (if possible) to distinguish a specific instance of a gadget from the generic class of gadget. Requesting bounds for these variables immediately makes the system slightly easier to use (as opposed to requesting bounds for potentially hundreds of variables once the specification is complete). Moreover, the system need these bounds and must ensure that users provide them, as good bounds decrease the likelihood of an unbounded solution to the constraint set when types are to be produced.

Gadget instances may be connected to other gadget instances on the canvas via edge placement. Edge placement not only imposes equality constraints on the ports at the tail of the edge and at the head of the edge, but for the sake of readability, all references to the head variable in the constraint set (and in the UI) are replaced with the tail variable.

As a user places gadget and/or edges, he or she will see the direct effects of these changes to the constraint set presented in the bottom frame of the user interface. As is reflected in the interface, the constraint set is stored with respect to the gadget that introduced the constraint (or rather, the gadget with which a specific constraint is associated) such that if a gadget is removed from the topology, the corresponding constraints may be removed as well. This is also necessary to distinguish constraints from the head to the tail in the event that an edge is deleted and the variables need to be re-separated.

Beyond the stand-alone, direct definition and specification of gadget classes, these 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 gadget) 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 need to be removed manually (*i.e.*, 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 gadget (*i.e.*, complex or unwieldy topologies) and for the promotion of reuse for commonly needed gadget instances. For example, a user may easily create a 3-way merge by connecting two merge gadget and exporting an envelope of that model.

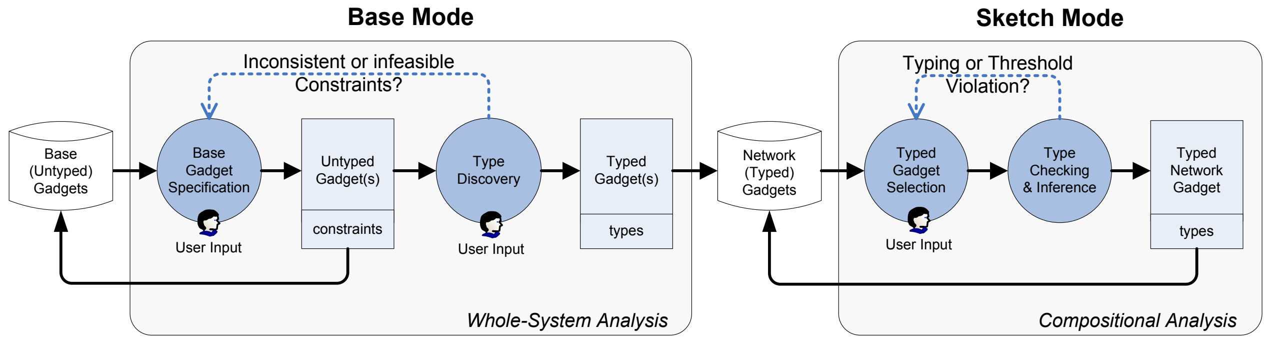


Figure 3. A process overview of Network modelling and assessment with NetSketch.

## C. *Manipulating (Typed) Network Gadgets*

Using this mode of operation involves the placement and connection of *Typed* gadgets (and holes) into a Typed Network. This enables a user to construct networks (*i.e.*, interconnected, typed gadgets) and to explore their connectivity and their interface properties. The presentation and use of the simplified type-centric (interval) interface allows a user to immediately see the range of valid inputs into a network of typed gadgets and yet safely ignore the complete set of constraints when connecting to other Typed gadgets or networks. Again, we refer to a Typed Network as a collection of Typed Gadgets that may contain holes, and trivially a typed gadget is an instance of a typed network that just lacks holes; as such the Typed Interface ultimately involves sketching (composing) Typed Gadgets and Typed Networks into larger Typed Networks.

To use an untyped gadget in typed mode, the NetSketch tool performs an automated whole-system analysis on the gadgets to produce types. While in the analysis phase, the constraint solver will process the current constraint set and will assign feasible bounds (if they exist) to constituent gadgets. If the user specifies specific optimization criteria (*e.g.*, maximize a specific edge, verify that flow is conserved) when converting an untyped gadget to a typed one, given that optimization constraint, the solver will attempt to produce a feasible range with respect to that constraint. 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 typed 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 in and of itself is exemplary of the benefit of the 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.

## D. *Transitioning from Untyped to Typed Modes*

An important "decision point" for NetSketch users is the determination of the point at which the user should abandon whole-system (exact) analysis and switch to compositional (sketched) analysis (*i.e.*, transition from untyped to typed modes of oepration). 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 (*i.e.*, node), to combine all gadgets into a single typed gadget, or to make groupings of untyped gadgets into typed gadgets).

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, this is clearly not necessary. The typed network (compositional analysis) interface loses some degree of specificity, so the time of transition and related granularity should be chosen carefully. The correct choice for such a fine granularity would be to experiment with placing different gadgets (or gadget holes) in place of other existing/defined gadgets.

Moreover, at some point the constraint sets in a topology of untyped gadgets may get sufficiently complex such that compositional analysis becomes the *preferred* (if not only) possibility for analysis. We define this point as the constraint *threshold*. The constraint threshold may be determined in any number of ways that might be beneficial to the user (*e.g.*, 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). Presently, our implementation of NetSketch is able to deal only with linear constraints, and as such, at this time this threshold is is left as a manual switch to be selected by the user.[5]

While not supported at this time, once an initial bounds analysis has been performed, it would technically be possible to export a typed gadget back to an untyped gadget. A typed gadget can be viewed as an untyped gadget with an extremely simplified constraint set, where the constraints would be simple interval inequalities that reflect the types discovered in analysis. This scenario is analogous to downcasting in programming languages.

[5]Automatically determining a value for this threshold given the shape/complexity of a constraint set (and the use of various non-linear programming libraries) is planned.

## E. *Manipulating Holes*

A hole is a placeholder for any unknown or under-specified gadget or network. Holes enable modeling and verificaion 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 be substituted.

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. A hole might be used by the user as a placeholder for a network for which he or she has partial information, or an entirely unknown network (or series of networks). The type system then can be used to specify the valid range of values for that unknown network 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 automate the searching for and the suggestion of 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.

## F. *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. To solve the constraint sets that are built as a result of the composition of modules in the GUI, we invoke the GNU Linear Programming Toolkit (GLPK) [3], which can be used to solve a system of constraints for linear programming and mixed integer programming problems. The decision to use GLPK is based on project maturity, community support and API availability. At present we emit our constraints in the GNU MathProg language and solve constraints on the local client machine. Moving forward, we intend to solve constraints remotely (using the client for a graphical interface only) and may leverage a project such as the Optimization Services project that defines an XML schema for optimization problems, optimization solutions, and facilitates the remote invocation of a solver to consume one and produce the other. Such a move would also be related to our desire to support a wider range of constraints (*i.e.*, not just linear) in the module composition phase.

## IV. 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].[6]

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).

## V. 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. This means that 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 should be open in the morning with the goal of ensuring that no backups occur within the city center, with the 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 each bridge could have open while ensuring no backups.

*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

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

single node (call it $\mathbf{F}$) for which the formal definition is:

$$\begin{aligned} N \;&= \{\mathbf{F}\}, \mathsf{In} = \langle 1 \rangle, Q = \varnothing, \mathsf{Out} = \langle 2, 3 \rangle \\ \mathsf{Par} \;&= \{1 \mapsto v_1 \cdot d_1, 2 \mapsto v_2 \cdot d_2, 3 \mapsto v_3 \cdot d_3\} \\ \mathsf{Con} \;&= \mathsf{Con}_{\text{nodes}} \cup \mathsf{Con}_{\text{links}} \qquad \text{where} \\ &\quad \mathsf{Con}_{\text{nodes}} = \{\text{regulating traffic through } \mathbf{F}\} \\ &\quad \mathsf{Con}_{\text{links}} \;= \{\text{lower/upper on } v_1, d_1 \ldots\} \end{aligned}$$

This definition specifies the structure of the junction, with the constraint set Con only containing predefined constraints that are required for all *fork* gadgets:

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

Constraint (1) enforces *conservation of density* when $\mathbf{F}$ 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 $\mathsf{Con}_{\text{links}}$ can be used to specify 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.

It is worth noting that other meaningful constraints could be introduced into the set of constraints 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 $\mathbf{M}$. The gadget is specified by the following six-tuple (omitted details and justifications are similar to those in Example 1):

$$\begin{aligned} N \;&= \{\mathbf{M}\}, \mathsf{In} = \langle 1, 2 \rangle, Q = \varnothing, \mathsf{Out} = \langle 3 \rangle \\ \mathsf{Par} \;&= \{1 \mapsto v_1 \cdot d_1, 2 \mapsto v_2 \cdot d_2, 3 \mapsto v_3 \cdot d_3\} \\ \mathsf{Con} \;&= \mathsf{Con}_{\text{nodes}} \cup \mathsf{Con}_{\text{links}} \qquad \text{where} \\ &\quad \mathsf{Con}_{\text{nodes}} = \{\text{regulating traffic through } \mathbf{M}\} \\ &\quad \mathsf{Con}_{\text{links}} \;= \{\text{lower/upper bounds on } v_1, d_1 \ldots\} \end{aligned}$$

*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 $\mathbf{X}$, defined as follows:

$$\begin{aligned} N \;&= \{\mathbf{X}\}, \mathsf{In} = \langle 1, 2 \rangle, Q = \varnothing, \mathsf{Out} = \langle 3, 4 \rangle \\ \mathsf{Par} \;&= \{1 \mapsto v_1 \cdot d_1, 2 \mapsto v_2 \cdot d_2, 3 \mapsto v_3 \cdot d_3, 4 \mapsto v_4 \cdot d_4\} \\ \mathsf{Con} \;&= \mathsf{Con}_{\text{nodes}} \cup \mathsf{Con}_{\text{links}} \qquad \text{where} \\ &\quad \mathsf{Con}_{\text{nodes}} = \{\text{regulating traffic through } \mathbf{X}\} \\ &\quad \mathsf{Con}_{\text{links}} \;= \{\text{lower/upper bounds on } v_1, d_1 \ldots\} \end{aligned}$$

The constraints regulating traffic through $\mathbf{X}$ 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:

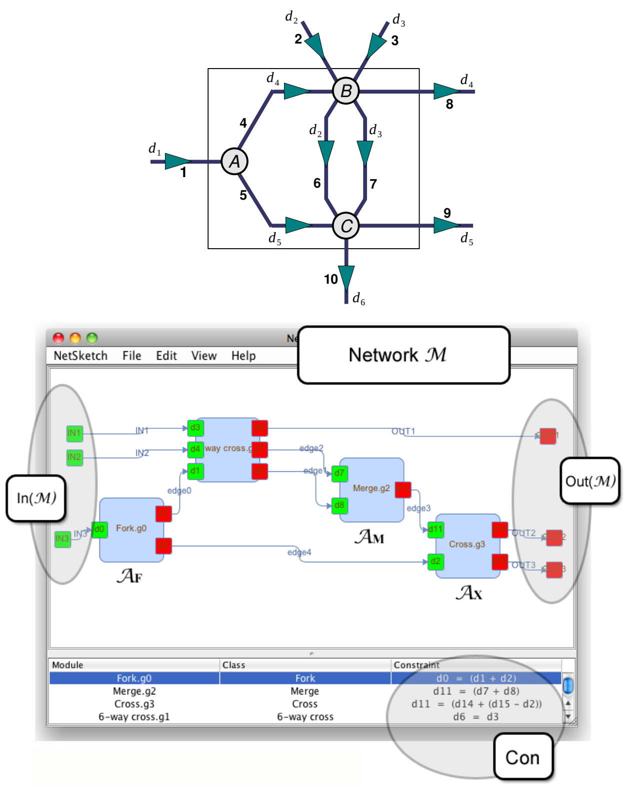
$$d_1 + d_2 \leqslant 10$$

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.



| Module | Class | Constraint |
|---|---|---|
| Fork.g0 | Fork | d0 = (d1 + d2) |
| Merge.g2 | Merge | d11 = (d7 + d8) |
| Cross.g3 | Cross | d11 = (d14 + (d15 - d2)) |
| 6-way cross.g1 | 6-way cross | d6 = d3 |

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

*Example 4:* This example is more complicated than the preceding ones in this section. Consider our engineer modelling 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 $\mathcal{M}$) capture this problem which builds on the previous three example gadgets. $\mathcal{M}$, consists of three nodes (gadgets) named $A, B, C$ and has 10 links named $1, 2, \ldots, 10$ ($N = \{A, B, C\}, \mathsf{In} = \langle 1, 2, 3\rangle, Q = \{4, 5, 6, 7\}, \mathsf{Out} = \langle 8, 9, 10\rangle$).

To complete the (untyped) specification of $\mathcal{M}$, we need to supply a function Par (assignment of parameters to all links in $\mathsf{In}^\times \cup Q \cup \mathsf{Out}^\times$) and a set Con (constraints over these parameters). For simplicity in this example, we restrict attention to density parameters, ignoring velocity parameters. For the sake of brevity, rather than provide a formal definition of Par (as in the previous examples), gadget $\mathcal{M}$, with every link assigned a density parameter, is illustrated in Figure 4.

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 (bottom). The instance of $B$ shown is a user-defined gadget, while $C$ which is shown as a single node in Figure 4 (top) 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 shown 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 (specific to these gadget instances) 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.[7] 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 and the tool will indicate what range of densities would be expected to enter that hole in the event that $d_6$ is maximized.

## VI. 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 set of 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 defined as follows:

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

where $t_i$ indicates the time of the $i^{th}$ frame and $t_f$ is the time of the last frame (it is assumed 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

[7]For a more complete treatment of what it means for a type to be invalid, weakly valid, strongly valid or optimal, we refer the reader to our companion paper [1].

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 is also wasteful of resources as 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).

Here we note that for a given stream, multiple $(r, s, w)$ values may exist, underscoring 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 which 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 be interested in performing on a video stream (or an aggregate thereof) at the server is that of 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.

Notice that the constraints associated with the various gadgets presented above are meant to exemplify safe constraints that could be asserted by a programmer or system integrator. Needless to say these constraints could be refined and/or made "tighter" (*i.e.*, more permissive). Moreover, by introducing additional "variables" (*e.g.*, delay or loss rates), the gadgets could be made more truthful to specific implementation details, *e.g.*, using concepts from Network Calculus to establish relationships between flows. Needless to say, the simplicity of the constraints we use is for ease of presentation and not a reflection of the capabilities of NetSketch and its DSL to deal with more elaborate constraint sets. Indeed, NetSketch's ability to deal with constraints at multiple levels of details (a.k.a., sketchiness) is one of NetSketch's salient features.

## VII. Related Work

Most previously proposed systems for reasoning about the behavior of distributed programs (Process algebra [4], Petri nets [5], Π-calculus [6], finite-state models [7], [8], [9], and even model checking [10], [11]) rely upon the retention of relatively high degrees of detail about the *internals* of a system's components in order to assess their interactions with other components. While this certainly affords these systems great expressive power, that expressiveness necessarily carries with it a burden of complexity. Such an approach is inherently unmodular in its analysis, in at least two ways. First, because details are not easily added to or shed from a representation or model when it is compared and interfaced with another, the specification of components must be highly coordinated for global analysis to be possible; the specifications are often wedded to particular methodologies and not sufficiently *general* to be amenable to multiple analysis approaches and interaction with differently-specified systems. Second, it is not generally possible to analyze portions of a system independently and then, without reference to the internals of those portions, assess whether they can be assembled together.

The capabilities to construct, model, infer, and visualize networks and properties of network constraints provided by NetSketch are similar to the capabilities and interfaces provided by modelling and checking tools such as Alloy [12]. Unlike Alloy, that can be used to model and check constraints on sets and relations, NetSketch focuses on providing more specific capabilities involving directed graphs.

One of the essential functionalities of NetSketch is the ability to reason about, and find solution ranges that respect, sets of constraints that happen to describe properties of a network. In its most general form, this is known as the *constraint satisfaction problem* [13] and is widely studied [14]. NetSketch types are linear constraints, so one variant of the constraint satisfaction problem relevant to our work is when the constraints under consideration are all linear. Finding solutions to linear constraints is a classic problem that has been considered in a large variety of work over the decades. There exist many documented algorithms [15, Ch. 29] and analyses of practical considerations [16]. However, the typical approach is to consider a homogenous list of constraints of a particular class. A distinguishing feature of NetSketch and the underlying formalism 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 – namely, NetSketch types, which are linear constraints that are varyingly restricted to make the analysis tractable – in exchange for a potentially more restrictive solution range, and the conjunction of large constraint sets is made more tractable by employing compositional techniques.

The work in this paper extends and generalizes our earlier work in TRAFFIC (*Typed Representation and Analysis of Flows For Interoperability Checks* [17]), and complements our earlier work in CHAIN (*Canonical Homomorphic Abstraction of Infinite Network protocol compositions* [18]). CHAIN and TRAFFIC are two distinct generic frameworks for analyzing existing grids/networks, and/or configuring new ones, of local entities to satisfy desirable global properties. Relative to one particular global property, CHAIN's approach is to reduce a large space of sub-configurations of the complete grid down to a relatively small and equivalent space that is amenable to an exhaustive verification of the global property using existing model-checkers. TRAFFIC's approach uses type-theoretic notions to specify one or more desirable properties in the form of invariants, each invariant being an appropriately formulated type, which are preserved when interfacing several smaller subconfigurations to produce a larger subconfiguration. CHAIN's approach is top-down, TRAFFIC's approach is bottom-up.

NetSketch leverages a rigorous formalism for the specification and verification of desirable global properties while remaining ultimately lightweight. By "lightweight" we mean to contrast our work to the heavy-going formal approaches – accessible to a narrow community of experts – which are permeating much of current research on formal methods and the foundations of programming languages (such as the work on automated proof assistants [19], [20], [21], [22], or the work on polymorphic and higher-order type systems [23], or the work on calculi for distributing computing [24]). In doing so, our goal is to ensure that the formalisms presented to NetSketch users are the *minimum* that they would need to interact with, keeping the more complicated parts of these formalisms "under the hood".

## VIII. Current and Future Work

NetSketch's current constraint system for untyped gadgets (which is limited to linear constraints) is intended to be a proof-of-concept to enable our work on typed networks (holes, types and bounds). We intend to expand the current constraint set that is supported within the tool to include more complex constraints. Closely related to this is the need to assess systems of non-trivial (*i.e.*, non-linear) constraints automatically for the sake of determining the *threshold* automatically. Finally, with respect to constraint satisfaction, we intend to decouple the solver from the client to enable much greater model processing power (*e.g.*, running on multiple machines in the cloud) accessible from lightweight clients.

As indicated in the accompanying paper on the NetSketch formalism [1], there is no natural ordering of types for sketches. When no optimal constraint function is provided, conversion from untyped gadget to a typed gadget may produce multiple different valid types. Types can be considered optimal based on the size of their value ranges (*e.g.*, a larger input range and smaller output range is preferable, as is typically the case with type systems), yet multiple "optimal" typings may exist. The algorithm that we use to assign types should be amended to try to establish a weight for various valid typings.

## References

[1] A. Bestavros, A. Kfoury, and A. L. M. Ocean, "Safe Compositional Network Sketches: Formalism," CS Dept., Boston University, Tech. Rep. BUCS-TR-2009-028, September 29 2009. [Online]. Available: http://www.cs.bu.edu/techreports/2009-028-scnsFormalism.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] C. A. Petri, "Communication with Automata," Ph.D. dissertation, Univ. Bonn, 1966.

[6] 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.

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

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

[9] ——, "Forward and backward simulations – part II: Timing-based systems," *Information and Computation*, vol. 128(1), pp. 1–25, Jul. 1996.

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

[11] 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.

[12] D. Jackson, "Alloy: a lightweight object modelling notation," *Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002. [Online]. Available: citeseer.ist.psu.edu/jackson01alloy.html

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

[14] E. Tsang, "A glimpse of constraint satisfaction," *Artif. Intell. Rev.*, vol. 13, no. 3, pp. 215–227, 1999.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, ser. The MIT Electrical Engineering and Computer Scienece Series. The MIT Press, McGraw-Hill Book Company, 1990.

[16] R. Fletcher, *Practical methods of optimization; (2nd ed.)*. New York, NY, USA: Wiley-Interscience, 1987.

[17] 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

[18] 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. [Online]. Available: http://www.cs.bu.edu/fac/best/res/papers/icnp03.pdf

[19] L. C. Paulson, *Isabelle: A Generic Theorem Prover*. Springer-Verlag, 1994, vol. LNCS 828.

[20] H. Herbelin, "A $\lambda$-calculus structure isomorphic to Gentzen-style sequent calculus structure," in *"Proc. Conf. Computer Science Logic"*, ser. LNCS, vol. 933. Springer-Verlag, 1994, pp. 61–75. [Online]. Available: http://coq.inria.fr/ herbelin/publis-eng.html

[21] A. Ciaffaglione, "Certified reasoning on real numbers and objects in co-inductive type theory," Ph.D. dissertation, Dipartimento di Matematica e Informatica Università di Udine, Italy, 2003, available as outline.

[22] K. Crary and S. Sarkar, "Foundational certified code in a metalogical framework," in *Nineteenth International Conference on Automated Deduction*, Miami, Florida, 2003.

[23] *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Paris, France, Jun. 2007.

[24] G. Boudol, "The $\pi$-calculus in direct style," in *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997, pp. 228–241.