

Safe Compositional Network Sketches: NetSketch Tool Implementation*

Nate Soule Azer Bestavros Assaf Kfoury Andrei Lapets

February 9, 2011

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 discuss a first implementation of the NetSketch system. We begin with a brief introduction to the NetSketch concept and formalism, and then discuss the methodology behind the type generation process as well as the technical implementation of the overall system. In a companion paper [1], 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 INTRODUCTION

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 holistic manner. For example, the development of applications with predictable timing properties necessitated 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, 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

*This work is supported in part by NSF awards CNS-0952145, CCF-0820138, CSR-0720604, CNS-1012798, and EFRI-0735974.

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 specificity 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. Similarly, 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 Section IV we look at the design decisions behind the type generation algorithms used to support compositional analysis. Section V covers the technical implementation details of the system. We conclude the paper with Section VI, a summary of current and future work for this implementation.

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 NetSketch (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 contrast *whole-system* and *compositional* analyses according to Figure 1, where $\llbracket x \rrbracket$ 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

<u>Gadgets</u>	<u>Whole-system</u>		<u>Compositional analysis</u>
A	$\llbracket A \rrbracket$	=	$\llbracket A \rrbracket$
$A \otimes B \otimes C$	$\llbracket A \otimes B \otimes C \rrbracket$	=	$\llbracket A \rrbracket \star \llbracket B \rrbracket \star \llbracket C \rrbracket$
$A \otimes \langle _ \rangle \otimes C$	$\llbracket A \otimes \langle _ \rangle \otimes C \rrbracket$?? $\stackrel{?}{=}$		$\llbracket A \rrbracket \star \llbracket \langle _ \rangle \rrbracket \star \llbracket C \rrbracket$
$A \otimes B' \otimes C$	$\llbracket A \otimes B' \otimes C \rrbracket$	=	$\llbracket A \rrbracket \star \llbracket B' \rrbracket \star \llbracket C \rrbracket$
...

Figure 1: Contrasting whole-system and compositional analyses.

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

¹A 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.

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 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 $*$ 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 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 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 nodes 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 tuples 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 $\mathcal{M}: (\text{In}_1, \text{Out}_1)$ and $\mathcal{N}: (\text{In}_2, \text{Out}_2)$ 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 is defined by rules of the

form shown in Figure 2.

$$\text{CONNECT}^{\square}: \frac{\Gamma \vdash \mathcal{M}:(In_1, Out_1) \quad \Gamma \vdash \mathcal{N}:(In_2, Out_2)}{\Gamma \vdash \mathcal{M} \square \mathcal{N}:(In_1, Out_2)}$$

where if $Out_1 = \langle \sigma_1, \dots, \sigma_n \rangle$ and $In_2 = \langle \tau_1, \dots, \tau_2 \rangle$ then $\sigma_1 <: \tau_1, \dots, \sigma_n <: \tau_n$

$$\text{LET}' : \frac{\Gamma \vdash \mathcal{M}_1:(In_1, Out_1) \cdots \Gamma \vdash \mathcal{M}_n:(In_n, Out_n) \quad \Gamma, \mathcal{X} : (In', Out') \vdash \mathcal{N}:(In, Out)}{\Gamma \vdash \text{Let } \mathcal{X} \in \{\mathcal{M}_1, \dots, \mathcal{M}_n\} \text{ in } \mathcal{N}:(In, Out)}$$

where $In_1 = \dots = In_n = In'$ and $Out_1 = \dots = Out_n = Out'$

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

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 \mathcal{X} of a network \mathcal{N} , we can place at will any of n different networks $\{\mathcal{M}_1 \dots, \mathcal{M}_n\}$ 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 underlies NetSketch. Collectively, they define the formal syntax of NetSketchs 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 $\mathcal{M} : (In, Out)$ be a typed specification for network \mathcal{M} , where In is an assignment of types to the input links and Out an assignment of types to the output links:

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

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

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.

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 tradeoffs is one of the main design goals of NetSketch.

III THE NETSKETCH TOOL

The NetSketch formalism has been partially implemented by the current version of the NetSketch tool. The NetSketch tool offers users the ability to visually create and define modules, and create connections among them. Subsets of connected gadgets (networks) may then be selected for inclusion in the type generation process. This paper describes the state of the tool as of its first release which captures many of the core features of NetSketch, but leaves others to future implementations. Section VI describes the functionality yet to be built. The tool can be found under Projects → NetSketch at the following URL: <http://www.cs.bu.edu/groups/ibench/>.

Figure 3 shows a screen of the NetSketch tool in action. Depicted are three gadgets from the domain of vehicular traffic: a merge, a fork, and a 2-way cross intersection. The screen of the tool is divided into two main areas, with a menubar running along the top edge. The middle section of the screen represents the canvas, onto which users will place modules and create connections between these modules to create networks. The bottom section of the screen presents the details of the currently selected gadget.

A user defines a network topology by creating new modules. This can be accomplished by creating a new module from scratch (*i.e.*, no ports or constraints defined), or by selecting from a library of pre-defined gadgets. Gadgets from the library come pre-built with a set number of ports (input or output variables), and a base set of linear constraints describing the gadget’s operational requirements. Both blank, and library gadgets can then be extended by adding, deleting, and modifying ports and constraints.

Ports are only given meaning when included in the constraints of the gadget containing the port. Thus port creation is inferred during constraint definition. As a user creates a new constraint, $x+y = z$ for example, the system performs syntactic analysis of the constraint to determine its variables, and automatically updates the list of ports for the gadget. As constraints are created, modified, and removed, the available ports for the given gadget will be added or removed as appropriate. Once a

³The 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.

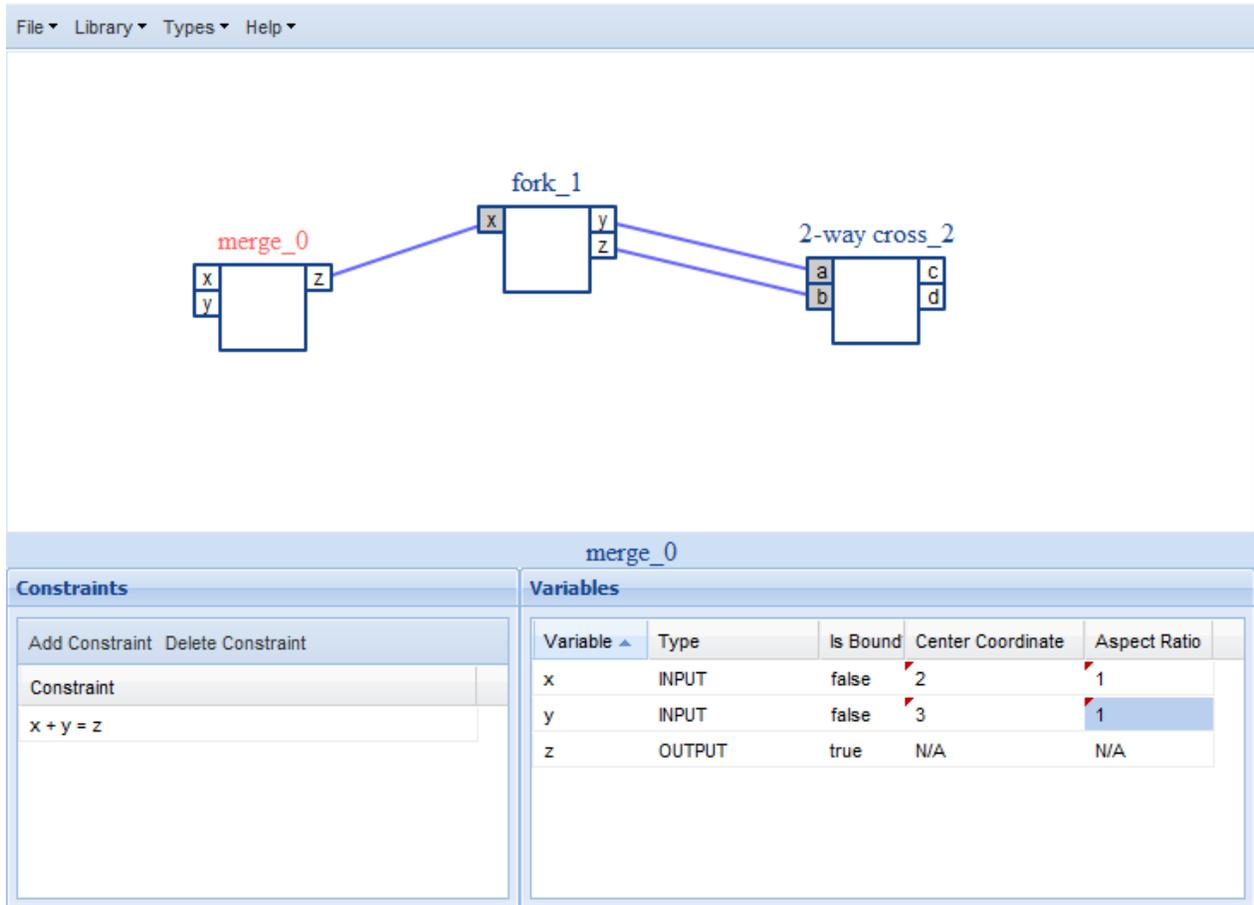


Figure 3: View of 3 connected gadgets in the NetSketch tool.

port is defined, it must be classified as either an input or an output port⁴. Classifying a port as an input or output causes it to be drawn on the canvas. Input ports align to the left of a gadget, and output ports to the right.

Once constraints are defined, and ports classified a gadget is ready for interfacing with other gadgets. The gadgets can be visually dragged around the canvas to allow for appropriate positioning in relation to other gadgets with which potential connections exist or to indicate logical groupings/relations. To connect two gadgets a user creates a line by dragging from the port of one gadget to the port of another (or among ports on the same gadget to create a loop). If port P_1 is connected to port P_2 then either P_1 is an input port, or P_2 is an input port, but not both (*i.e.*, an exclusive-or relationship).

Once two ports are connected their binding status in the **Variables** area of the screen is updated from *false* to *true*. In addition the screen visually indicates this with a line between the ports, and the receiving end of the connection shaded gray. Though not represented explicitly in the **Constraints** area of the screen, an implicit constraint is created for every port connection. An equality constraint $P_n = P_m$ is implied for every connection of port P_n to port P_m .

⁴In future implementations the ability to have internal variables that are neither input or output will be allowed.

As only intra-gadget constraints are displayed on the screen at any given time, variable names need not be unique across gadgets. Internally NetSketch performs variable renaming by prepending the gadget name to the variable name. From the user’s perspective only the gadget specific variable name (*i.e.*, x , not $\text{fork}_1.x$) is displayed. This is possible and safe since the system guarantees unique gadget names through a global counter added to each gadget name.

When a connected set of modules is in a stable state the user can choose to generate a type for that set. By selecting an option from the menubar a type generation window will open. This window, as shown in Figure 4, allows the user to select among the available modules. A type can be created for a single module if the user determines a typed gadget is easier to manipulate and use than a untyped gadget, or a subset of connected modules may be collectively typed. The decision regarding the level of granularity in type generation is an important one. This represents the point where exact analysis is replaced with compositional analysis.

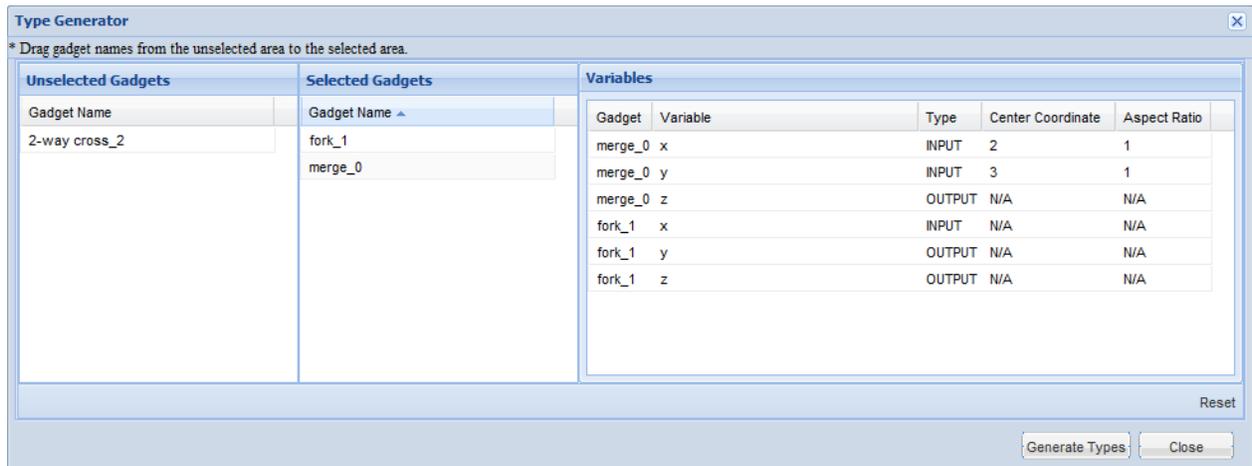


Figure 4: Type generation window with two connected gadgets selected for type creation.

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) method 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 left as a manual decision to be made by the user.

In its current form the NetSketch tool will provide the generated type to the user, but for informational purposes only; this information is not retained in the system. In future implementations the tool would transition from *base* mode to *sketch* mode at this point. In *sketch* mode all gadgets would be typed, and compositional analysis would be possible. This is where the tangible benefits of NetSketch are realized, and as such this will be the primary area of development for future implementations. See section VI for more details.

The types generated for a set of gadgets are non-empty intervals over \mathbb{R} . For each non-connected port P exposed within the set of gadgets being typed, an interval of the form $P: [P_{min}, P_{max}]$ will be generated. Optimal typings of this form can not be guaranteed to be uniquely generated for the

input variables without further guidance from the user. In this implementation this guidance takes the form of a center point and an aspect ratio relating all *input* variables. See section IV for the reasons behind this requirement and the details of the center-point/aspect-ratio solution.

With types generated, what were formerly potentially complex and numerous constraints are now simple intervals that can be viewed, composed, and analyzed efficiently. In addition, with this level of typing, unknowns in the topology can easily be left as *holes* that can have their typings inferred without further specification simply by connecting them appropriately to defined gadgets. These features follow naturally from the NetSketch formalism, and will be added to future implementations.

IV TYPE GENERATION

Generating types from sets of untyped gadgets involves transforming linear constraints into intervals over \mathbb{R} . This process is divided into two high level steps: input port type generation, and output port type generation. As the output type generation can use the results of the input types to create more accurate results, these sub-processes are performed in order as listed above.

Input Port Type Generation In order to generate types for the input ports of a set of gadgets, it helps to visualize the set of linear constraints that define the set as a convex hull. Figure 5 shows such a hull in 2-space (*i.e.*, for a set of constraints over two input variables). Here we see 4 constraints labeled Constraint 1 through Constraint 4. The convex hull formed by their intersection defines the set of feasible input values.

To create intervals for the input variables we need to find a largest enclosed hyper-rectangle within the convex hull. Such an area is not necessarily unique. Various options exist for techniques to select a single typing from among these non-unique hyper-rectangles. On the more expressive and accurate side options exist such as selecting all or some subset of the optimal enclosed hyper-rectangles and defining the type to be the union of such as set. For this implementation, a more restrictive process was used that involves defining a center point for the hyper-rectangle, along with an aspect ratio relating all input variables. In Figure 5 the center point (x, y) is displayed along with the aspect ratio relating x to y .

Given a center point and an aspect ratio, a unique maximally enclosed hyper-rectangle can be identified given the set of linear constraints for the gadgets. Intuitively this can be visualized (in 2 or 3-space) as enlarging a hyper-rectangle that begins as a single point at the given center point, in increments defined by the given aspect ratio, until the hyper-rectangle intersects with the convex hull defined by the linear constraints of the gadget set. Programmatically this is accomplished by determining the set of diagonals defined by the hyper-rectangle (labeled Diagonal 1, and Diagonal 2 in Figure 5). There exist 2^{n-1} such diagonals for an n -dimensional hyper-rectangle. Given the center point and aspect ratio of the desired hyper-rectangle, expressions describing the diagonals can be created trivially in parametric form (which the system later converts to the standard linear equation form for use with an existing linear programming solver). With these diagonals defined, the closest intersection (to the center point) with the given linear constraints is then located using linear programming. Four of the eight potential intersection points in Figure 5 are highlighted with circles. Once the closest intersection point $I_{x,y}$ is identified a hyper-rectangle of dimensions $|I_x - C_x|$ by $|I_y - C_y|$ centered at $C_{x,y}$ can be defined. The bounds of this hyper-rectangle on any given axis represent the bounds of the interval for that axis's variable. This example was given in 2-space for

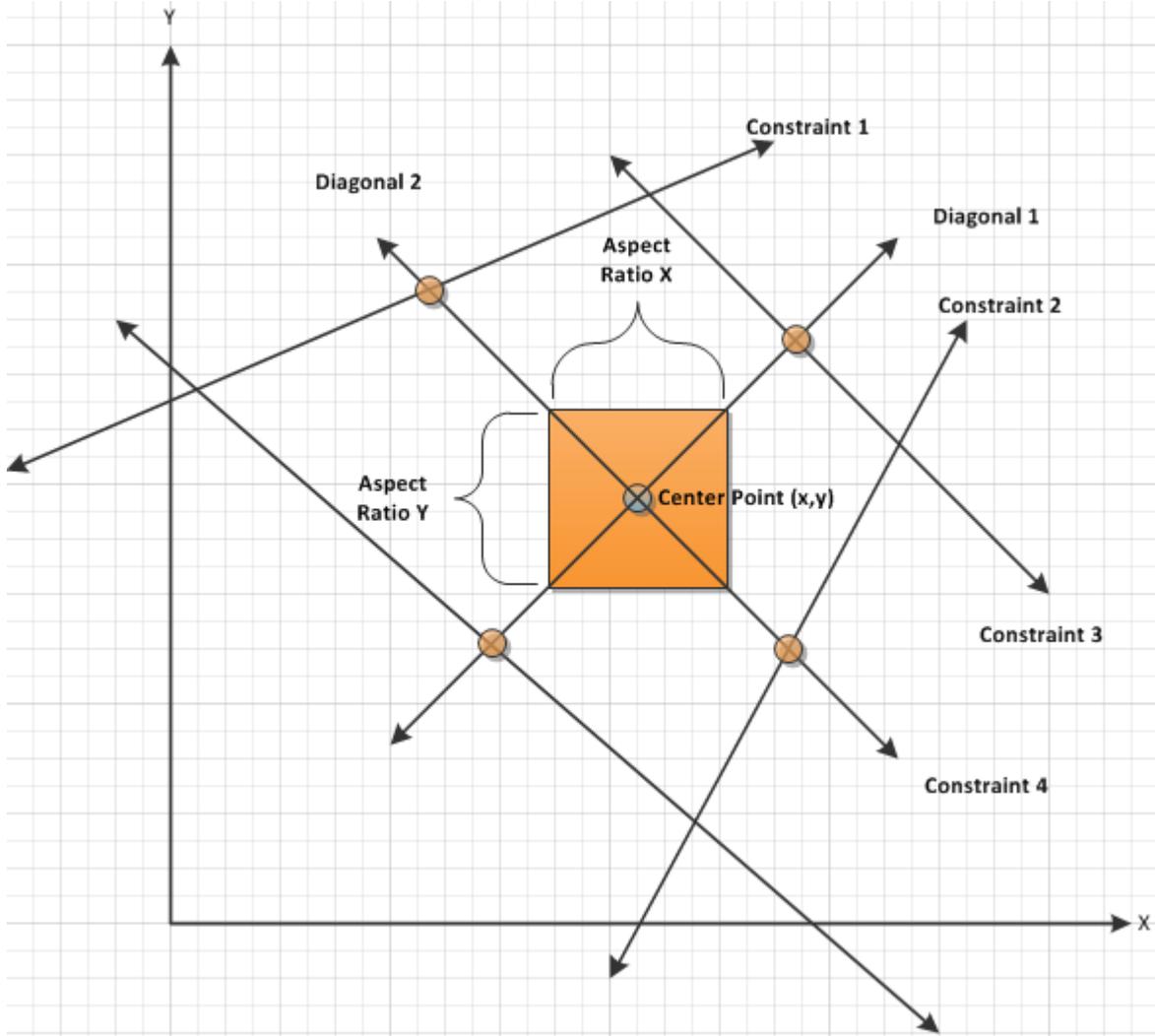


Figure 5: Input Type Generation

visual clarity, but the principles extend to n dimensions where $n \geq 2$ (special case coding exists to handle $n = 1$).

The discussion to this point has assumed that we already have the set of linear constraints to use when generating the input type. It must be noted, however, that the set of linear constraints defined by the user does not equal the set used for these constraints. This is the case for two reasons. Firstly, the set of linear constraints defined by the user does not explicitly contain the equality constraints restricting connected ports between gadgets to equal each other. These constraints are implied in the visual connections drawn between gadgets, but made explicit in the inner workings of the NetSketch tool. Secondly, when specifying the set of linear constraints for a given gadget, the user may well define constraints relating the input and output ports. The generation of the maximally enclosed hyper-rectangle as described above requires the constraints to be restricted to only contain variables from the input ports. To accommodate this need the NetSketch tool first performs a projection of the given constraints, plus the implicit connection constraints, onto only those dimensions representing

the input variables. For example, given input ports $I = \{a, b, c\}$, output ports $O = \{x, y, z\}$, and a set of linear constraints C over $I \cup O$, the system will project C onto the 3-dimensional space of I . The resulting constraint set is used in the generation of the maximally enclosed hyper-rectangle.

Output Port Type Generation As with input type generation it is helpful to visualize the linear constraints as forming a convex hull as depicted in Figure 6. To determine the feasible output values, unlike the maximally enclosed hyper-rectangle needed for input ports, a minimally enclosing hyper-rectangle must be identified. The determination of this hyper-rectangle is significantly simpler than for that of its input counterpart. Here an optimal enclosing is unique, so center point and aspect ratios are not required.

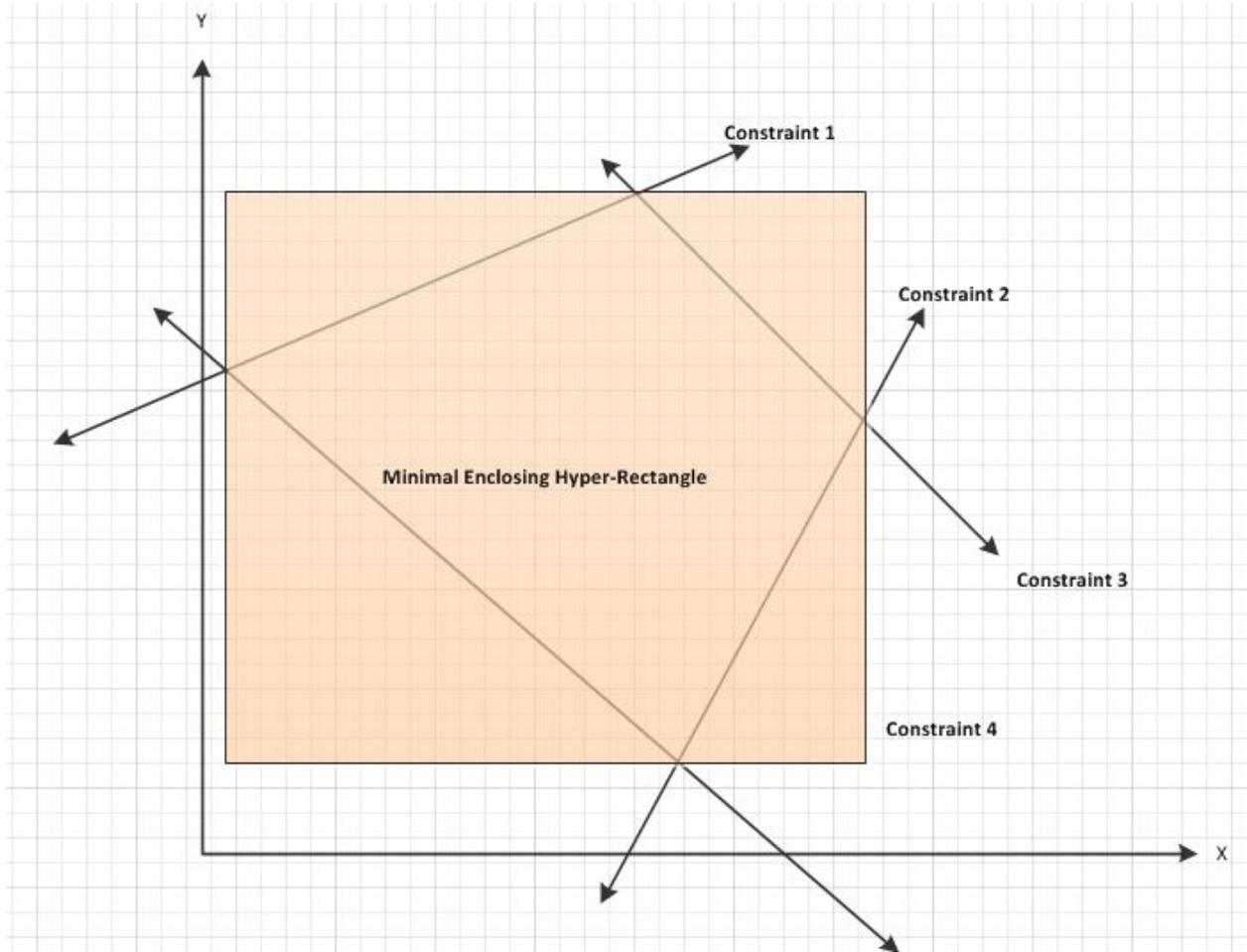


Figure 6: Output Type Generation

The hyper-rectangle can thus be computed using linear programming to solve the system of equations first with an objective function of $\text{Maximize}(v)$, then again with an objective function of $\text{Minimize}(v)$ for each output variable v . The solution resulting from maximizing v will become the upper bound for the variable's type, and the solution for minimizing v will become the lower bound (*i.e.* $\forall v \in \mathcal{I} : \text{type}(v) = [\text{Solution}_{Min}, \text{Solution}_{Max}]$).

As mentioned previously, the constraints used when calculating the output types should include those generated as the input types. The intervals created during input type generation are therefore converted into simple linear constraints (*e.g.*, $x : [0, 100]$ becomes two constraints: $x \geq 0$, and $x \leq 100$). These constraints are then added to the original constraints for use in determining the output types. Without these extra constraints, the result would be correct, but the range of values for the output types would be wider than they truly need to be as the valid input values will have been restricted during conversion to intervals in all but the most pathological cases.

V TECHNICAL IMPLEMENTATION

The NetSketch tool architecture comprises a client component and a server component as represented by the **User Interface** and **Core Engine** boxes respectively in Figure 7.

The client-server paradigm was employed to allow for a lightweight web deployment, while still retaining a non-browser-resident server component for the linear programming and other computationally heavy tasks. The client and server communicate over HTTP using Ajax-style requests.

User Interface The user interface was built using pure JavaScript and HTML. Standalone executables offering graphical user interface capabilities were considered (Java, Python), but ultimately a web based solution was decided upon due to a desire for an easily accessible, easily updatable, zero-installation solution. While other web-based platforms (JavaFX, Silverlight, Air, Flash) contain more robust graphical capabilities, it was determined that JavaScript/HTML alone could provide the required GUI capabilities and would avoid attaching the project to a heavyweight proprietary framework.

In order to alleviate some of the burden of ensuring cross browser compatibility, and development of a rich set of widgets, the ExtJS JavaScript Framework[11] was employed to provide the basic GUI elements. ExtJS is an open source framework that provides a wide array of user interface components as well as JavaScript utilities for DOM (document object model) manipulation, and a simple AJAX model.

In addition to ExtJS, JSGL (the JavaScript Graphics Library)[9], a pure JavaScript vector graphics toolkit, was used. JSGL provided the vector graphics capabilities needed to draw widgets, their ports, and the connections between them. JSGL, as with ExtJS, also servers to hide cross browser incompatibilities.

Core Engine The core of the NetSketch tool is implemented as a server side component. The server is written in Haskell, with much of the heavy mathematical processing being delegated to external C-based modules. The main executable makes use of the Happstack Web Framework[4]. NetSketch uses the built in HTTP server functionality of Happstack to expose the NetSketch API over the web. HTTP GET requests can be constructed to provide the NetSketch server with the description of the gadgets (including ports, connections, and constraints) in a format based on the domain specific language defined in the work outlining the NetSketch formalism[1].

Once the HTTP server component has received a request it is passed to the *untyped language engine* for parsing. The *untyped language engine* parses the request based on the NetSketch untyped language DSL, and passes the text representing linear constraints to the *constraint language engine*. The grammars for both the NetSketch untyped DSL, and the linear constraint language are defined in annotated BNF. The Haskell parser generator Happy[6] was used to generate parsers based on

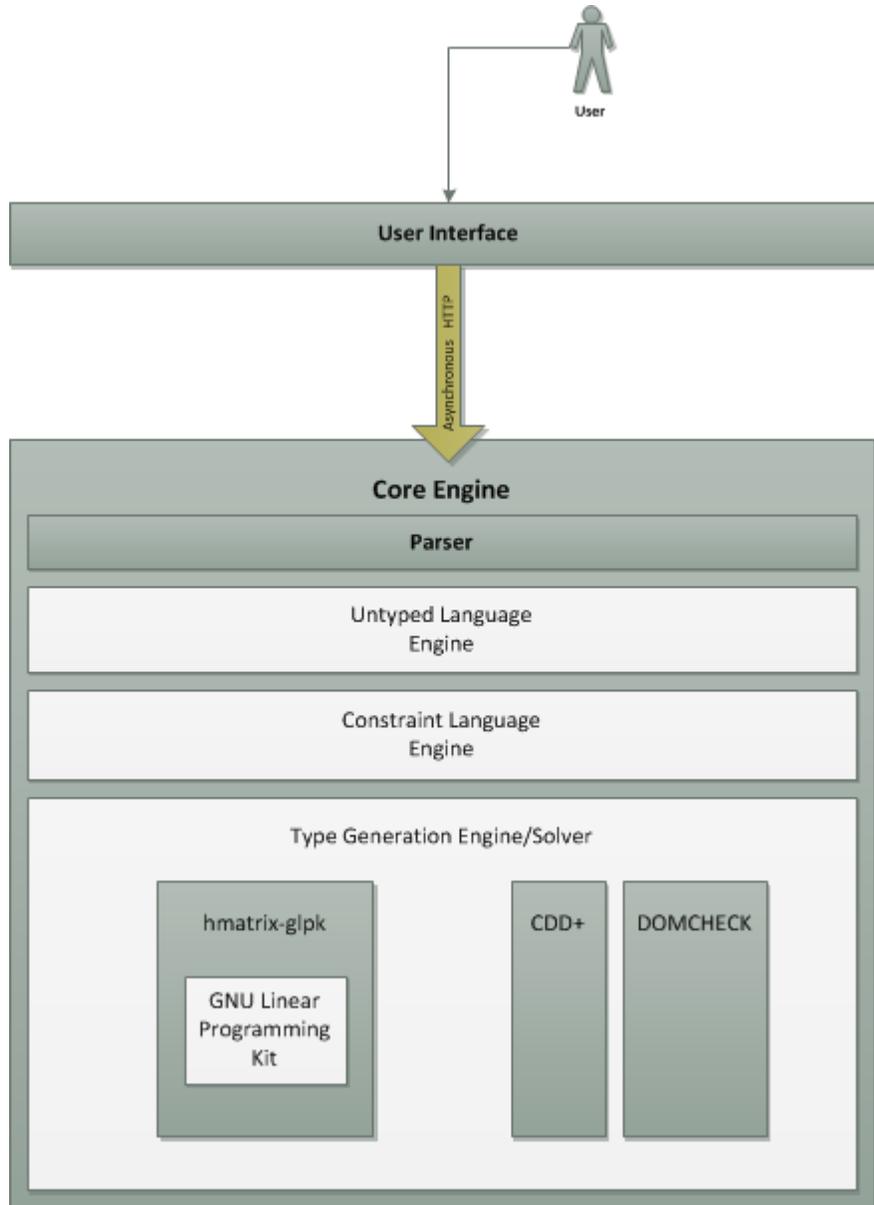


Figure 7: Architecture of NetSketch Tool

these grammars. Beyond the parsing functionality, each language engine provides functionality related to the manipulation of its respective language (*e.g.*, simplifying and removing redundancy from linear constraints).

After a successful parse, the structure representing the network topology described in the request is sent to the *type generation engine*. This module first performs input type generation, followed by output type generation. Input type generation must first project the constraints onto a subset of the original dimensions (specifically those corresponding to the input ports). This projection is done using two external C/C++-based modules: CDD+[5] and Domcheck[8]. CDD+ is a C++ implementation of the Double Description Method for vertex and extreme ray enumeration. Domcheck

is a program that computes minimal linear descriptions of projections of polytopes. These modules are distributed as C/C++ source code. The only modifications made were to DOMCHECK in order to allow non-interactive execution (i.e. to call in batch without a user present).

Both the output type generator, and the input type generator (after projection) make use of linear programming techniques to identify boundaries of the generated types. The linear programming is accomplished via a Haskell wrapper, `hmatrix-glpk`[10], around the GNU Linear Programming Kit (GLPK)[3]. The GLPK is a C-based callable library providing routines for linear programming, mixed integer programming, and other related problems. NetSketch makes use of the GLPK’s implementation of the Simplex method. HMatrix-GLPK provides a pure Haskell interface to this and a select set of other features from GLPK.

VI CURRENT AND FUTURE WORK

A number of planned areas of future work exist both related to filling out the functionality of the NetSketch tool to match the power expressed in the NetSketch formalism, and in expanding on the formalism itself.

Tool Completion In its current state the NetSketch tool implements only a subset of the features and functionality that are expressed in the formalism defined in [1]. The tool will be expanded to allow for the transition to *sketch* mode where the types generated will be persistently attached to the sets of gadgets they relate to. These typed gadgets will then be able to be analyzed, and various scenarios tested for satisfaction of safety constraints. In addition the concept of a network *hole* will be introduced into the tool. This idea, defined in detail in the work outlining the formalism[1], allows for the creation of nodes within the topology that contain unknown constraints. Types for these *holes* can then be inferred based on their connections to the rest of the network. Currently variables within constraints must be classified as input, or output variables. In future implementations, internal variables will be allowed that do not correspond to ports of the gadget.

Two other tool limitations exist currently. These correspond to issues when using the external C libraries to project the constraints onto only the input dimensions. These issues may stem from bugs in the 3rd party libraries or in the method of use of these libraries by NetSketch. Their root causes have not yet been determined. The first corresponds to an issue where the same set of equations is producing two different outputs depending upon the order of the variables (the ordering is consistent within a set of equations for a particular run of the system). For example given a set of equations of the form $a_{1i}, a_{2i}, a_{3i}, a_{4i} = c_i$ where a_{1i} through a_{4i} represent the coefficients of variables 1 through 4 of the i th equation, and c_i represents a constant for the i th equation, re-ordering a_{1i} and a_{2i} consistently through all equations may produce different results than the original ordering. The temporary workaround for this issue is to add non-negativity constraints to all free variables. The second issue appears when a particular situation is reached, either through the nature of the system of equations being represented, or via the size of the system (as it is only seen in larger systems) a system core dump occurs in CDD. Further research on both issues will be conducted as time allows.

Formalism A deeper examination into the proper model for selecting among optimal typings could lead to an alteration of both the tool (in its current requirement for a center point and aspect ratio), and potentially in the formalism by allowing for an enhanced ability to express types as

unions of intervals or other constructs. Papers describing the formalism[1], as well as related papers [2] and [7] outline a number of additional ideas for furthering the core concepts behind NetSketch.

References

- [1] Azer Bestavros, Assaf Kfoury, Andrei Lapets, and Michael Ocean. Safe Compositional Network Sketches: Formalism. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2009. Tech. Rep. BUCS-TR-2009-029, October 1, 2009 - <http://www.cs.bu.edu/techreports/pdf/2009-029-netsketch-formalism.pdf>.
- [2] Azer Bestavros, Assaf Kfoury, Andrei Lapets, and Michael Ocean. Safe Compositional Network Sketches: Tool and Use Cases. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2009. Tech. Rep. BUCS-TR-2009-028, October 1, 2009 - <http://www.cs.bu.edu/techreports/pdf/2009-028-netsketch-tool.pdf>.
- [3] GNU Project Developers. GLPK GNU Project. <http://www.gnu.org/software/glpk/>, January 2011.
- [4] Matthew Elder and Jeremy Shaw. Happstack - A Haskell Web Framework. <http://happstack.com/index.html>, January 2011.
- [5] Komei Fukuda. cdd and cddplus homepage. http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html, January 2011. Swiss Federal Institute of Technology.
- [6] Andy Gill and Simon Marlow. Happy - The Parser Generator for Haskell. <http://www.haskell.org/happy/>, January 2011.
- [7] Andrei Lapets, Assaf Kfoury, and Azer Bestavros. Safe Compositional Network Sketches: Reasoning with Automated Assistance. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2010. Tech. Rep. BUCS-TR-2009-028, January 19, 2010 - <http://www.cs.bu.edu/techreports/pdf/2010-001-netsketch-aa.pdf>.
- [8] Francois Margot. Francois Margot Homepage. <http://wpweb2.tepper.cmu.edu/fmargot/>, January 2011. Carnegie Mellon.
- [9] Tomas Rehorek. JavaScript Graphics Library (JSGL) official homepage. <http://www.jsgl.org/doku.php>, January 2011.
- [10] Alberto Ruiz. HackageDB: hmatrix-glpk-0.2.1. <http://hackage.haskell.org/package/hmatrix-glpk>, January 2011.
- [11] Sencha. Sencha - Ext JS - Client-side Javascript Framework. <http://www.sencha.com/products/js/>, January 2011.