Safe Compositional Equation-based Modeling of Constrained Flow Networks*

Nate Soule[1] Azer Bestavros[1] Assaf Kfoury[1] Andrei Lapets[1]

[1]Department of Computer Science, Boston University, USA, {nsoule,best,kfoury,lapets}@bu.edu

Abstract

Numerous domains exist in which systems can be modeled as networks with constraints that regulate the flow of traffic. Smart grids, vehicular road travel, computer networks, and cloud-based resource distribution, among others all have natural representations in this manner. As these systems grow in size and complexity, analysis and certification of safety invariants becomes increasingly costly. The NetSketch formalism and toolset introduce a lightweight framework for constraint-based modeling and analysis of such flow networks. NetSketch offers a processing method based on type-theoretic notions that enables large scale safety verification by allowing for compositional, as opposed to whole-system, analysis. Furthermore, by applying types to the modeled networks, analysis of composite modules containing incomplete or underspecified components can be conducted. The NetSketch tool exposes the power of this formalism in an intuitive web-based graphical user interface. We describe the NetSketch formalism and tool, a translation from an instantiation of the NetSketch formalism to the equation-based modeling language Modelica, and the development of an accompanying Haskell library, HModelica, that enables the integration of NetSketch and the OpenModelica modeling platform.

Keywords Flow networks, Network analysis, Safety verification, Constraint based modeling

1. Introduction

Many large scale systems can be modeled as assemblies of subsystems, each of which produces, consumes, or regulates a flow. Such models can contain variables and constraints representing the safe operation of the system. Networks that may be represented in this manner cross many domains within software, hardware, electrical, material, structural and other areas. Electric grids, vehicular road networks, and computer networks are all modeled cleanly in this structure; in addition, so are less immediately obvious examples, such as the governance of service level agreements (SLAs) in cloud computing environments. In the case of SLAs, a physical processor may generate a flow that is regulated by schedulers and consumed by computing processes. In electric grids, power plants may act as nodes producing flow, with transmission lines, and transformers routing and regulating flow to commercial and residential customers (who may in turn act not only as sinks, but as sources when, for example, they have solar panels). Extended detail and further examples are described in separate papers [5, 22, 23]. Verification of safety invariants across such a system is a critical analysis task, but this task can quickly grow costly as the complexity of a model increases. The NetSketch formalism and accompanying tool offer a constraint-based modeling solution capable of handling such complexity while providing an efficient analysis engine.

The nodes in a constrained flow network may contain arbitrarily complex constraints that serve to connect them and regulate their operation. Solving for a set of feasible values for the variables of the system will produce the inputs and outputs that constitute "safe" usage. This is a desirable task both from a modeling perspective: ensuring or discovering the range of safe values, and from a design perspective: considering alternative "what if" scenarios and inspecting their properties in search of optimal values. In large systems the size and complexity of the set of constraints and variables under consideration can limit or even prohibit whole-system analysis. To allow for an analysis under these circumstances, NetSketch employs techniques from type theory to simplify the constraints of the network at various levels of the system's composition. A *type* is given to various subsets of the network under consideration. Each sub-network of nodes can then, for the purposes of analysis, be replaced with an opaque container that exposes only the ports at its interfaces. This new component is then regulated by a type at each of its ports. By considering only the types, and not the potentially complex set of internal constraints, it is possible to more efficiently analyze this new component in the context of the larger network, and to determine safe ways to connect this component to others during a design process.

In this paper we describe the NetSketch formalism and tool, and make connections between this work and the broader equation-based objected-oriented modeling do-

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

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.

EOOLT 2011 website: http://www.eoolt.org/2011/

$$\textsc{Hole}\quad \frac{(X, \mathsf{In}, \mathsf{Out}) \in \Gamma}{\Gamma \vdash (X, \mathsf{In}, \mathsf{Out}, \{\,\})}$$

$$\textsc{Module}\quad \frac{(\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})\ \text{module}}{\Gamma \vdash (\mathcal{B}, I, O, \{C\})} \qquad (\mathcal{B}, I, O, C) = {}'(\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})$$

$$\textsc{Connect}\quad \frac{\Gamma \vdash (\mathcal{M}, I_1, O_1, \mathcal{C}_1) \qquad \Gamma \vdash (\mathcal{N}, I_2, O_2, \mathcal{C}_2)}{\Gamma \vdash (\mathbf{conn}(\theta, \mathcal{M}, \mathcal{N}), I, O, \mathcal{C})} \quad \theta \subseteq_{\text{1-1}} O_1 \times I_2,\ I = I_1 \cup (I_2 - \mathsf{range}(\theta)),\ O = (O_1 - \mathsf{domain}(\theta)) \cup O_2,\ \mathcal{C} = \{C_1 \cup C_2 \cup \{\, p = q \,|\, (p,q) \in \theta \,\} \,|\, C_1 \in \mathcal{C}_1, C_2 \in \mathcal{C}_2\}$$

$$\textsc{Loop}\quad \frac{\Gamma \vdash (\mathcal{M}, I_1, O_1, \mathcal{C}_1)}{\Gamma \vdash (\mathbf{loop}(\theta, \mathcal{M}), I, O, \mathcal{C})} \quad \theta \subseteq_{\text{1-1}} O_1 \times I_1,\ I = I_1 - \mathsf{range}(\theta),\ O = O_1 - \mathsf{domain}(\theta),\ \mathcal{C} = \{C_1 \cup \{\, p = q \,|\, (p,q) \in \theta \,\} \,|\, C_1 \in \mathcal{C}_1\}$$

$$\textsc{Let}\quad \frac{\Gamma \vdash (\mathcal{M}_k, I_k, O_k, \mathcal{C}_k) \quad \text{for } 1 \leqslant k \leqslant n \qquad \Gamma \cup \{(X, \mathsf{In}, \mathsf{Out})\} \vdash (\mathcal{N}, I, O, \mathcal{C})}{\Gamma \vdash (\ \mathbf{let}\ X \in \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}\ \mathbf{in}\ \mathcal{N}\ , I, O, \mathcal{C}'\ )}$$

$$\mathcal{C}' = \left\{ C \cup \hat{C} \cup \{\, p = \varphi(p) \,|\, p \in I_k \,\} \cup \{\, p = \psi(p) \,|\, p \in O_k \,\} \,\middle|\, 1 \leqslant k \leqslant n,\ C \in \mathcal{C},\ \hat{C} \in \mathcal{C}_k,\ \varphi : I_k \to \mathsf{In},\ \psi : O_k \to \mathsf{Out} \right\}$$

Figure 1. Rules for Untyped Network Sketches.

main. In Section 2 we describe the domain specific language at the heart of NetSketch. In Section 3 we describe the NetSketch tool and its architecture. We take a deeper look at the type generation algorithms in Section 4. In Section 5 we investigate the relation of NetSketch to current equation-based object-oriented modeling solutions (Modelica in particular) via two avenues. First, we examine the use of Modelica to assist various computational tasks required by NetSketch; here, we introduce a Haskell library that exposes the power of Modelica to the NetSketch engine. Second, we define a translation from NetSketch models to Modelica that allows for whole system analysis of those models via simulation. Finally, we end with a discussion of related and future work in Section 6.

2. NetSketch Formalism

In a constrained flow network each node of the network or system may impose constraints on its inputs and outputs. The network and its entire constraint set form an exact model[1]. Any whole-system analysis of the network must compute the solution space of the constraint set for the given network. Our compositional approach uses types to approximate the constraints on the interface of each node or group of nodes. In this way sub-systems can be analyzed individually at an exact level, whereas the whole system can be analyzed based solely on the results of the subsystem analyses rather than the entire set of constraints. This method allows for efficient analysis of large systems even when the cost of a whole system analysis does not scale linearly with the size of the system. Further, the compositional aspect of this method allows for analysis to occur in cases where it otherwise would require more information *i.e.*, in incomplete systems. When a portion of the overall system has unknown constraints, but a known interface, NetSketch can infer the types that will allow safe operation of the system using the rest of the network and its connectivity to the incomplete "hole".

[1] Here by "exact" we mean with respect to those properties under consideration in the model. Any model is by neccessity an approximation of the system being represented.

The NetSketch formalism defines a domain specific language for describing constrained flow networks. In its original form [4] the DSL consists of five main constructs: *Module*, *Hole*, *Connect*, *Loop*, and *Let*. These are described below, and the corresponding rules for constructing network descriptions are depicted in Figure 1.

Module *Module* defines a new node in the network. This node is atomic *i.e.*, not composed of other nodes.

Hole A *hole* in a network describes an area that is incomplete (*e.g.*, not yet designed or unknown to the modeler). It provides the information that is known about this hole (only the number of inputs and outputs) without the need to fully specify the constraints. NetSketch enables its users to infer the minimal requirements to be expected of (or to be imposed on) such holes. This enables the design of a system to proceed based only on the promised functionality of missing parts.

Connect *Connect* allows for two distinct networks to be combined into a larger network. This construct binds a subset of the output ports of one network to a subset of the input ports of another. The result is a new network that can in turn be connected to others.

Loop *Loop* allows for the connection of an output port of a network to be connected to an input port of the same network.

Let *Let* is used to specify a set of networks that may be placed in a given network hole.

3. NetSketch Tool and Architecture

The NetSketch formalism is partially implemented by the current version of the NetSketch tool.[2] The NetSketch tool offers users the ability to visually create and define modules, and to create connections among them to form *network sketches*. Subsets of 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

[2] The tool can be found under Projects → NetSketch at the following URL: `http://www.cs.bu.edu/groups/ibench/`.

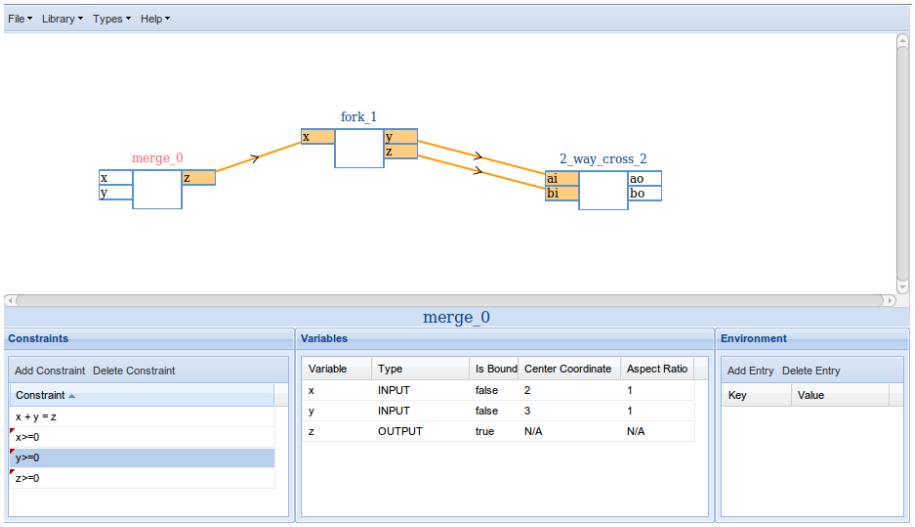

Figure 2. View of a network consisting of 3 connected modules in the NetSketch tool.

others to future implementations. Section 6 discusses some of the functionality yet to be added.

3.1 Interface and User Experience

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

Creating Modules A user can begin defining a network by first introducing new modules. This can be accomplished by creating a new module from scratch (*i.e.*, with no ports or constraints defined), or by selecting from a library of pre-defined modules and network sketches. Modules from the library come pre-built with a set number of ports (input or output variables), and a base set of linear constraints describing their operational requirements. Both blank and library modules can then be extended by adding, deleting, and modifying ports and constraints.

Ports are only given meaning when included in the constraints of the module 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 module. As constraints are created, modified, and removed, the available ports for the given module will be added or removed as appropriate. Once a port is defined, it must be classified as either an input or an output port[3]. Classifying a port as an input or output causes it to be drawn on the canvas. Input ports align to the left of a module, and output ports to the right.

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

Connecting Modules Once constraints are defined, and ports classified a module is ready for interfacing with other modules and networks. The modules can be visually dragged around the canvas to allow for appropriate positioning in relation to other modules with which potential connections exist or to indicate logical groupings/relations. To connect two modules a user creates a line by dragging from the port of one modules to the port of another (or among ports on the same module 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* and the screen visually indicates this with a line between the ports; an arrow indicates the direction of flow, and both ends of the connection are shaded. 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.

As only the constraints of a single module are displayed on the screen at any given time, variable names need not be unique across a network. Internally, NetSketch performs variable renaming by prepending the module name to the variable name. From the user's perspective only the module specific variable name (*i.e.*, `x, not fork_1.x`) is displayed. This is possible and safe because the system guarantees unique module names through a global counter added to each module name.

Generating Types 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 3, allows the user to select among the available modules. A type can be created for a single module if the user determines a typed version is easier to manipulate and use than an untyped one, or a subset of connected modules may be collectively typed. The decision regarding the level

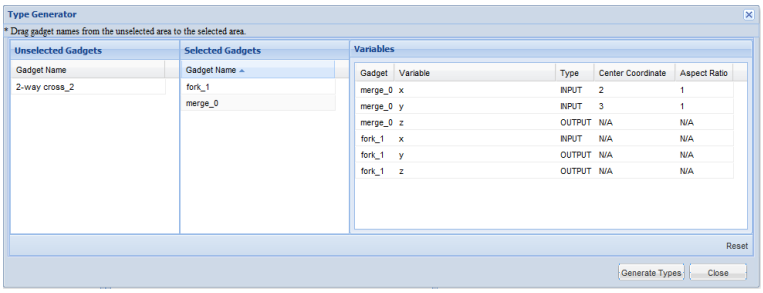

Figure 3. Type generation window with two connected modules selected for type creation.

of granularity in type generation is an important one. This represents the point where exact analysis is replaced with compositional analysis.

At some point the constraint sets in a network of untyped modules 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 nodes, number of connections, 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 leaves the decision regarding the value of this threshold to the user.

Once typed, a network is replaced visually by a single container - slightly shaded to differentiate it from untyped modules on the canvas. This process can be infinitely recrusive, in that a network of typed modules can itself be given a type. The constraints shown in the bottom portion of the screen are replaced with the intervals inferred for each port.

The types generated for a set of modules are non-empty intervals over $\mathbb{R}$. For each non-connected port P exposed within the set of modules 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 4 for the reasons behind this requirement and the details of the center-point/aspect-ratio solution, and Section 6 for a description of work underway to alleviate this need.

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 network can easily be left as *holes* that can have their typings inferred without further specification simply by connecting them appropriately to defined modules and networks. Holes can be created in a fashion similar to that of modules, with the exception that ports are listed explicitly as opposed to being inferred from the constraint set. The *Let* construct of the formalism, which describes which modules may be placed in a hole, is applied in the tool via the ability to select existing components from the library as potential hole replacements.

Persistence At any point the user can chose to save their canvas in a persistent form. The tool will convert the internal representation of the model into JavaScript Object Notation (JSON), and prompt the user to open or save the generated JSON file. Users can then later load their saved modules from disk to continue their modeling/analysis effort.

3.2 Architecture

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 4.

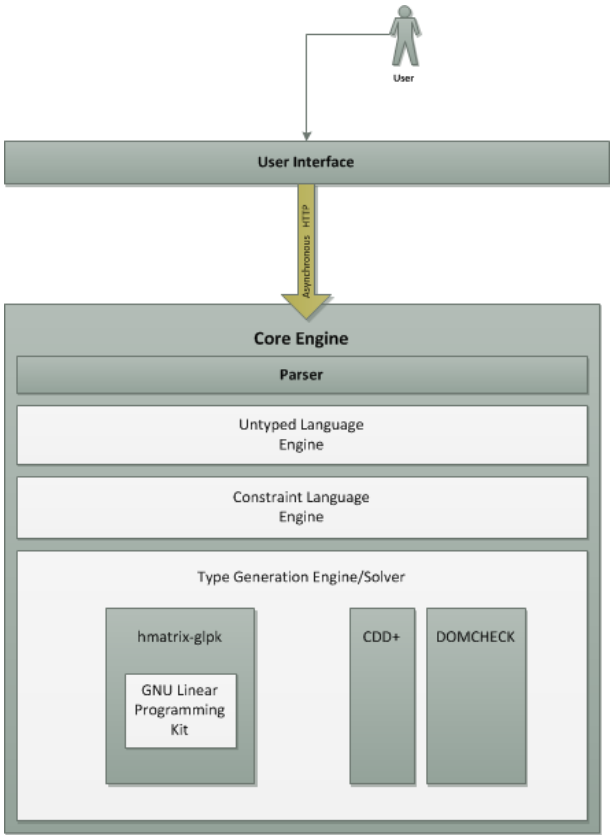

Figure 4. Architecture of NetSketch Tool

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 chosen 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 and 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 [21] 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) [19], 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, or to an implementation of the Modelica platform. The main executable makes use of the Happstack Web Framework [10]. 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 network (including ports, connections, and constraints) in a format based on the domain specific language defined in the work outlining the NetSketch formalism [4].

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 [12] was used to generate parsers based on 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 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+ [11] and Domcheck [15]. 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 can be accomplished via one of two mechanisms. The original implementation used a Haskell wrapper, hmatrix-glpk [20], around the GNU Linear Programming Kit (GLPK) [9]. 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. The other mechanism was developed after creating the HModelica Haskell library (see Section 5). In this method NetSketch makes calls via HModelica to an instance of the OpenModelica [17] platform. Here Modelica code is executed to perform the required linear programming tasks. By using OpenModelica 3, external libraries (hmatrix, hmatrix-glpk, and glpk) were no longer required, simplifying the code base.

4. Type Generation

Generating types from sets of untyped modules 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 the order listed above.

Input Port Type Generation In order to generate types for the input ports of a set of modules, 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 four 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[4] within the convex hull. Such an area is not necessarily unique. Various options exists 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 a subset of the possible enclosed hyper-rectangles and defining the type as their union. Work on the use of dynamic adaptive

[4] In this paper all hyper-rectangles are axis-aligned. For brevity we use the term "hyper-rectangle" to 'refer to "axis-aligned hyper-rectangle" throughout.

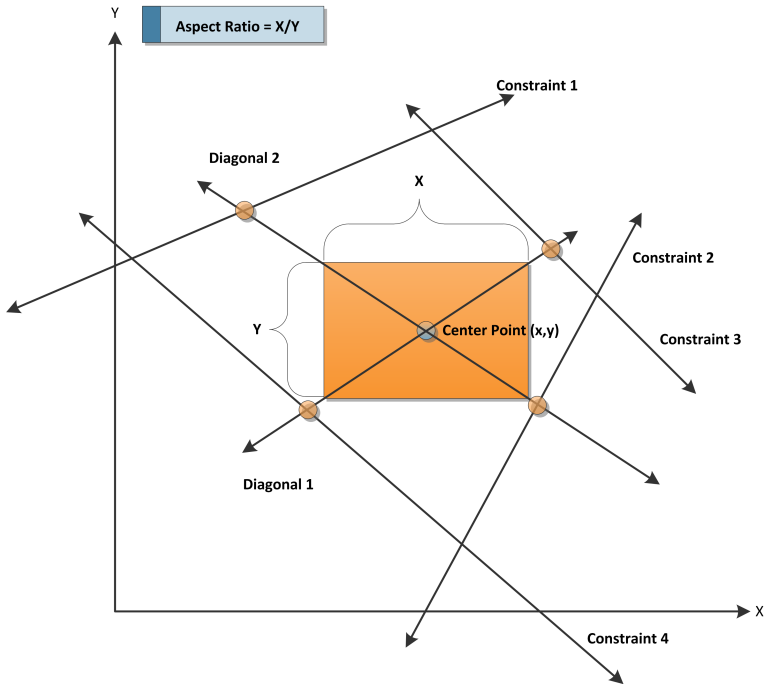

Figure 5. Input Type Generation

types is underway as described in Section 6. 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 modules. 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 module 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 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. First, the set of linear constraints defined by the user does not explicitly contain the equality constraints requiring connected ports between modules to equal each other. These constraints are implied in the visual connections drawn between modules, but made explicit in the inner workings of the NetSketch tool. Secondly, when specifying the set of linear constraints for a given module, 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: an optimal enclosing is unique, so a center point and aspect ratios are not required.

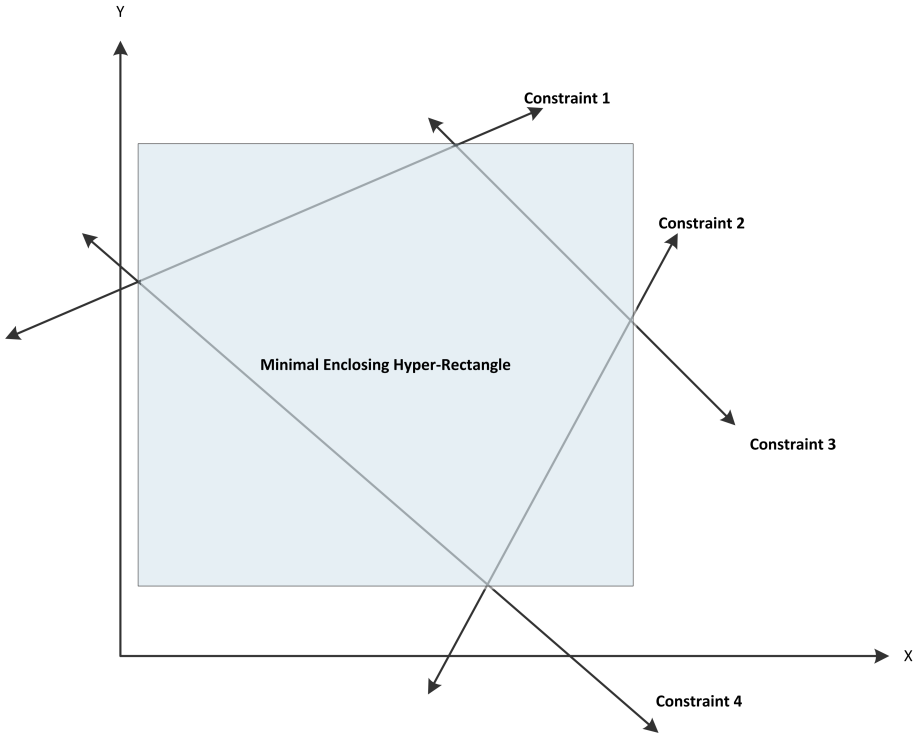

Figure 6. Output Type Generation

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

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: in all but the most pathological cases, the valid input values will have been restricted during conversion to intervals.

5. Harnessing Modelica

Well-established constraint-based modeling systems exist today. NetSketch shares a variety of similarities with these tools, but also bears numerous non-trivial differences. Notably, NetSketch in its current form does not explicitly consider time. Other constraint-based modeling tools, such as Modelica [2], are largely centered around time and use simulation over time as their main form of analysis. Some work has been done to show that a variation of NetSketch can be created to more natively incorporate the concept of time. Here, variables of the constraints are replaced by functions of the same name that accept a time variable as an argument. Given the simulation-based nature of Modelica and similar systems, other differences from NetSketch arise, such as the need for balanced systems over equations (rather than inequalities) [1].

Despite such differences, the overlap that does exist offers a great opportunity for various forms of integration. Here, we examine two forms of relation to Modelica: as a computation platform, and as an environment for working with translated NetSketch models.

5.1 Modelica as a Computation Platform

Modelica offers a wealth of functionality as well as a robust library. The extensive library provides both reusable models and reusable functions spanning many domains. This library can be of use to both NetSketch modelers (see sections 5.2 and 6), and to the NetSketch tool implementation itself.

Modelica and the functions defined in the Modelica library can be used directly by the NetSketch implementation as a processing engine. For example, the NetSketch engine requires frequent use of linear programming techniques, namely the simplex method. A function implementing this has been defined in Modelica code and can therefore be used by NetSketch to "farm out" some of the more mathematically heavy computations.

To gain access to the power of Modelica from within the NetSketch tool, a reusable Haskell library was developed to expose the functionality of the OpenModelica implementation to Haskell code. This library, HModelica, enables Haskell developers to create, manipulate, and simulate Modelica models, in addtion to directly executing func-

tions written in the Modelica language. Through the use of this library the NetSketch simplex code was replaced with calls to OpenModelica, alleviating the need for a handful of Haskell- and C-based libraries that previously were tasked with this work. Having a single platform and access mechanism for performing these types of tasks simplifies the NetSketch code base, and this impact will continue to grow as the set of tasks handed to Modelica increases.

The library exposes the OpenModelica API in two ways. The primary mechanism is in place for a subset of the OpenModelica API calls. These functions are implemented as type-safe calls with full translation to and from Haskell types. Second, for any functions not implemented in this manner (the number continues to decrease as development continues), a single function is implemented allowing the caller to send commands to Modelica as a string, and then to receive the results as a string. This allows for the execution of any arbitrary Modelica command.

HModelica has the potential to open Modelica up to the community of Haskell developers. As such, its use can extend outside of NetSketch. To that end the library is being added to the Haskell package repository HackageDB [8]. Here, it will be available for public download and use in the Cabal package format.

5.2 Translation to Modelica

Modelica and NetSketch share enough in common that a translation between the two can be defined. Here, we concentrate on the translation from NetSketch to Modelica; however, a subset of the models developed in Modelica (those with linear constraints) could be directly translated into typed NetSketch networks. This would provide NetSketch users with access to a wider array of pre-built components. A translation in this direction would map Modelica classes and related definitions to NetSketch module definitions with connections between classes and compositions of modules accomplished via NetSketch *Connect* and *Loop* constructs. A formal definition of such a mapping is being considered for future work, as discussed in Section 6.

The reverse direction, a translation from NetSketch to Modelica, generates models that can be used to perform simulation as a safety analysis tool. This process is outlined in detail in an accompanying work [22] and is described at a high level here. To accomplish a translation, two restrictions must be placed on the model during the process. First, any inequalities defined in the NetSketch constraints must be transformed to a form of validation check, as opposed to an active regulator of the system (as the equations section of a Modelica model must contain only that - equations). In some models this may require a binding of a subset of the variables involved in the constraints to specific values for a given simulation of the system (to allow the simulation to uniquely determine the flow). Second, the system must be balanced (not over- or underdetermined). This again may result in the binding of particular variables to concrete values for a given run of the simulation. In these cases single simulations can be run to test "what-if" scenarios corresponding to the particular binding given to the variables, or a set of simulations may be run on the extremes of the valid range of values for each given variable to determine a broader notion of safety across those ranges. Only the extremes of the intervals must be tested because the constraints in the current implementation are linear and thus form a convex hull; no gaps in safe ranges may exist.

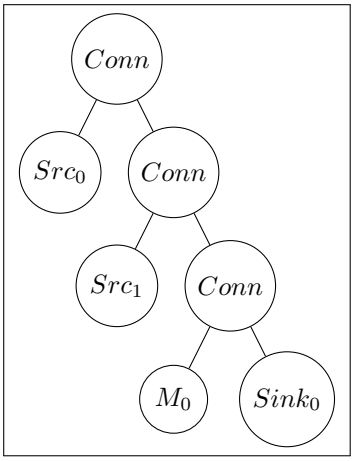

Figure 7. Tree view of the NetSketch network depicted in Figure 8

To reduce the number of variables that must be bound to concrete values, the NetSketch model is first analyzed to construct a minimal covering set. Such an analysis defines a set of variables $\mathcal{S}_{\mathrm{Min}} \subseteq I \cup O$ where I and O represent the set of inputs and outputs, respectively, of the system.

As an example consider Figure 8. Here 6 variables, a, b, c, d, e, f, and a constraint set exist to regulate flow within the system. Since M_0 conserves flow via the constraint $c + d = e$, we need only bind two variables, namely a, and b, to concrete values in order to determine the entire system. Since c, d, e, and f all depend on a and b to determine their values, these variables need not be considered when providing concrete values to drive a Modelica simulation.

An accompanying report [22] defines two algorithms for constructing $\mathcal{S}_{\mathrm{Min}}$. The first is quite efficient, involving two passes of the tree representing a NetSketch model (see Figure 7 for an example), but may not always produce the minimal set. It is causal in nature, and thus does not consider the potential positive impact of variables down the causal chain of the network. The first pass builds two transition relations, and the second actually constructs $\mathcal{S}_{\mathrm{Min}}$ using a set of formal rules and the transition relations from the first pass. A Haskell implementation of this process has been created and will be incorporated directly into the existing implementation as described in Section 6. The second algorithm described in [22] will always produce a minimal set, but has a worst-case exponential running time under a naive implementation. This algorithm transforms a system into a set of propositional logic implication statements representing how knowledge about one variable (or set of variables) implies knowledge about others. The problem is thus transformed into a search for the minimal number of propositional atoms that must be explicitly bound to $true$ in order to imply the conjunction of atoms representing all

variables in the system. A hybrid approach is also described that allows for the use of the first algorithm to set a maximum size of $\mathcal{S}_{\text{Min}}$ from which the second can start. This variant allows for significant savings in computation time.

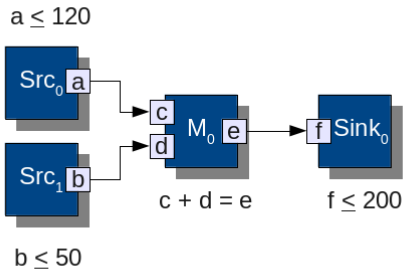

Figure 8. Two source modules, a merge, and a sink.

Once a minimal covering set is constructed, a translation can occur. This again involves a traversal of the tree representing the NetSketch model. Here, as each *Module*, *Hole*, *Conn*, *Loop*, and *Let* node is visited, an abstract representation of a Modelica model is incrementally constructed. NetSketch modules (and holes) are transformed into entities representing Modelica class definitions (or a restricted version thereof) with any equation-based constraints represented directly in the `equation` section of the resulting class definition. For all variables in $\mathcal{S}_{\text{Min}}$ the Modelica `parameter` modifier is used. Inequality constraints are moved to a "driver" class created to organize the system and provide validation checks that the model is safe. Within the driver class all modules/holes are present as instances of their respective classes. Appropriate initial value equations for the variables in $\mathcal{S}_{\text{Min}}$ are present with user-specified bindings. Modelica `connect` statements are used where NetSketch *Connect* and *Loop* constructs existed. The driver is thus a flat representation of the network. The driver also contains a single additional boolean variable, not present in the initial model: *isValid*. This variable is set to equal the conjunction of all the inequality constraints that existed in the individual modules/classes (as these could not be included in the `equation` sections of their owning classes). In this way a user can examine this variable post-simulation to determine if the model is safe under the given parameters. The resulting abstract representation is then transformed into a string which can be written to a text file, or sent directly to Modelica via the HModelica interface described above.

6. Related and Future Work

This work extends and generalizes our work in TRAFFIC [3], and complements our earlier work in CHAIN [7]. An essential functionality of NetSketch is the ability to reason about, and find solution ranges that respect, sets of constraints. In its general form, this is the widely studied *constraint satisfaction problem*. NetSketch types are linear constraints, and linear constraint satisfaction is a classic problem for which many documented algorithms exist. 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. Other formalisms and methods, such as [16], seek to enable early detection of problems in a model by applying types to constraint sets in a modular way, but are intended for providing assurances that compilation prior to analysis/simulation will succeed. In contrast the use of types in NetSketch directly support the analysis of the model itself.

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 [18, 13], or the work on calculi for distributing computing [6]). 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".

A number of planned areas of future work exist related to extending the functionality of the NetSketch tool to more closely match the power expressed in the NetSketch formalism, furthering integration with existing equation-based modeling tools, and extending the formalism itself.

Tool Enhancements Network *Holes* can currently be used with the *Let* construct to select elements from the library to act as hole replacements. Future versions of the tool will allow for selection from additional sources (the current canvas, persisted models, etc). 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 module.

NetSketch models the direction of data flow explicitly (*i.e.* ports are marked as either input or output). By default in Modelica's acausal system this is not the case. While NetSketch requires all ports to be causal, bidirectionality can be modeled through either the use of two connections - each representing a direction of flow, or by allowing flow across a single connection to be either positive or negative. Connecting an output port to an input port in NetSketch requires the former be a subtype of the latter. This imples that bidirectional flow over a single connection would require the participating ports have identical types. The NetSketch formalism allows for both of these methods of modeling bidirectionality, though extensions to the current implementation may make such modeling more accesible and transparent. Single connection bidirectionality may benefit, for example, from the extension of the current system's strictly linear constraints to include constructs such as the absolute value function.

Tool Integration As described in Section 5, Modelica offers a wealth of reusable components. Formally defining a translation from a Modelica model to NetSketch would

allow NetSketch users to quickly make use of the breadth of components developed for the Modelica platform. The translation is restricted in the current implementation to a simplified subset of models with linear equations. It should be noted, however, that the restriction to linear constraints is an artifact of the implementation, and not the formalism. The NetSketch formalism is parameterized by the chosen constraint space, and thus allows for a much more general set of constraints than the current tool implements.

The algorithms defined in Section 5 will be integrated into the current tool implementation to allow in-tool exports of NetSketch models to Modelica models. Direct execution of the resulting models will also be implemented as a function of the tool.

Formalism A deeper examination of the proper model for selecting among optimal typings is currently underway and will likely lead to an alteration of both the tool (in its current requirement for a center point and aspect ratio), and potentially of the formalism. Enhancements to the type system to allow for the expression of types as unions of intervals or as function of the state of the network connections is being explored. In addition, work is currently being undertaken on a version of the formalism that restricts the constraints to a particular subset of linear equations resulting in a simplified type inference mechanism, and an expanded set of tractable forms of analysis, while still allowing for an expressive constraint language with real-world applicability. Papers describing the formalism [4], as well as related papers [5, 14] outline a number of additional ideas for furthering the core concepts behind NetSketch.

References

[1] Modelica Association. Modelica Language Specification 3.2. Technical report, Modelica Association, 2010. http://www.modelica.org/documents/ModelicaSpec32.pdf.

[2] Modelica Association. Modelica and the Modelica Association. https://www.modelica.org/, May 2011.

[3] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Ibrahim Matta. Typed Abstraction of Complex Network Compositions. In *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP'05)*, Boston, MA, November 2005.

[4] 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.

[5] 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.

[6] Gérard Boudol. The π-calculus in direct style. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pages 228–241, 1997.

[7] Adam Bradley, Azer Bestavros, and Assaf 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.

[8] Hackage Community. Hackagedb. http://hackage.haskell.org, May 2011.

[9] GNU Project Developers. GLPK GNU Project. http://www.gnu.org/software/glpk/, January 2011.

[10] Matthew Elder and Jeremy Shaw. Happstack - A Haskell Web Framework. http://happstack.com/index.html, January 2011.

[11] Komei Fukuda. cdd and cddplus homepage. http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html, January 2011. Swiss Federal Institute of Technology.

[12] Andy Gill and Simon Marlow. Happy - The Parser Generator for Haskell. http://www.haskell.org/happy/, January 2011.

[13] Hugo Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *"Proc. Conf. Computer Science Logic"*, volume 933 of *LNCS*, pages 61–75. Springer-Verlag, 1994.

[14] 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.

[15] Francois Margot. Francois Margot Homepage. http://wpweb2.tepper.cmu.edu/fmargot/, January 2011. Carnegie Mellon.

[16] Henrik Nilsson. Type-based structural analysis for modular systems of equations. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, July 2008.

[17] Open Source Modelica Consortium (OSMC). Welcome to OpenModelica. http://www.openmodelica.org/, May 2011.

[18] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume LNCS 828. Springer-Verlag, 1994.

[19] Tomas Rehorek. JavaScript Graphics Library (JSGL) official homepage. http://www.jsgl.org/doku.php, January 2011.

[20] Alberto Ruiz. HackageDB: hmatrix-glpk-0.2.1. http://hackage.haskell.org/package/hmatrix-glpk, January 2011.

[21] Sencha. Sencha - Ext JS - Client-side Javascript Framework. http://www.sencha.com/products/js/, January 2011.

[22] Nate Soule, Azer Bestavros, Assaf Kfoury, and Andrei Lapets. Safe Compositional Equation-based Modeling of Constrained Flow Networks. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2011. Tech. Rep. BUCS-TR-2011-014, June 5, 2011.

[23] Nate Soule, Azer Bestavros, Assaf Kfoury, and Andrei Lapets. Use Cases for Compositional Modeling and Analysis of Equation-based Constrained Flow Networks. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2011. Tech. Rep. BUCS-TR-2011-019, July 5, 2011.