

Safe Compositional Equation-based Modeling of Constrained Flow Networks¹

Nate Soule* Azer Bestavros* Assaf Kfoury* Andrei Lapets*

*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 Appendix A, and in separate papers [5, 22]. 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 domain. 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

¹This work is supported in part by NSF awards CNS-0952145, CCF-0820138, CSR-0720604, CNS-1012798, and EFRI-0735974.

HOLE	$\frac{(X, \text{In}, \text{Out}) \in \Gamma}{\Gamma \vdash (X, \text{In}, \text{Out}, \{ \})}$	
MODULE	$\frac{(\mathcal{A}, \text{In}, \text{Out}, \text{Con}) \text{ module}}{\Gamma \vdash (\mathcal{B}, I, O, \{C\})}$	$(\mathcal{B}, I, O, C) = (\mathcal{A}, \text{In}, \text{Out}, \text{Con})$
CONNECT	$\frac{\Gamma \vdash (\mathcal{M}, I_1, O_1, C_1) \quad \Gamma \vdash (\mathcal{N}, I_2, O_2, C_2)}{\Gamma \vdash (\mathbf{conn}(\theta, \mathcal{M}, \mathcal{N}), I, O, C)}$	$\theta \subseteq_{1-1} O_1 \times I_2, I = I_1 \cup (I_2 - \text{range}(\theta)), O = (O_1 - \text{domain}(\theta)) \cup O_2,$ $C = \{C_1 \cup C_2 \cup \{p = q \mid (p, q) \in \theta\} \mid C_1 \in \mathcal{C}_1, C_2 \in \mathcal{C}_2\}$
LOOP	$\frac{\Gamma \vdash (\mathcal{M}, I_1, O_1, C_1)}{\Gamma \vdash (\mathbf{loop}(\theta, \mathcal{M}), I, O, C)}$	$\theta \subseteq_{1-1} O_1 \times I_1, I = I_1 - \text{range}(\theta), O = O_1 - \text{domain}(\theta),$ $C = \{C_1 \cup \{p = q \mid (p, q) \in \theta\} \mid C_1 \in \mathcal{C}_1\}$
LET	$\frac{\Gamma \vdash (\mathcal{M}_k, I_k, O_k, C_k) \text{ for } 1 \leq k \leq n \quad \Gamma \cup \{(X, \text{In}, \text{Out})\} \vdash (\mathcal{N}, I, O, C')}{\Gamma \vdash (\mathbf{let } X \in \{\mathcal{M}_1, \dots, \mathcal{M}_n\} \mathbf{in } \mathcal{N}, I, O, C')}$	$C' = \{C \cup \hat{C} \cup \{p = \varphi(p) \mid p \in I_k\} \cup \{p = \psi(p) \mid p \in O_k\} \mid 1 \leq k \leq n, C \in \mathcal{C}, \hat{C} \in \mathcal{C}_k, \varphi : I_k \rightarrow \text{In}, \psi : O_k \rightarrow \text{Out}\}$

Figure 1. Rules for Untyped Network Sketches.

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. A discussion of related and future work is presented in Section 6. Finally, in Appendix A, the NetSketch to Modelica translation is examined in depth.

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 model². 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 sub-system 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”.

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.³ 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 others to future implementations. Section 6 discusses some of the functionalities yet to be added.

²Here by “exact” we mean with respect to those properties under consideration in the model. Any model is by necessity an approximation of the system being represented.

³The tool can be found under Projects → NetSketch at the following URL: <http://www.cs.bu.edu/groups/ibench/>.

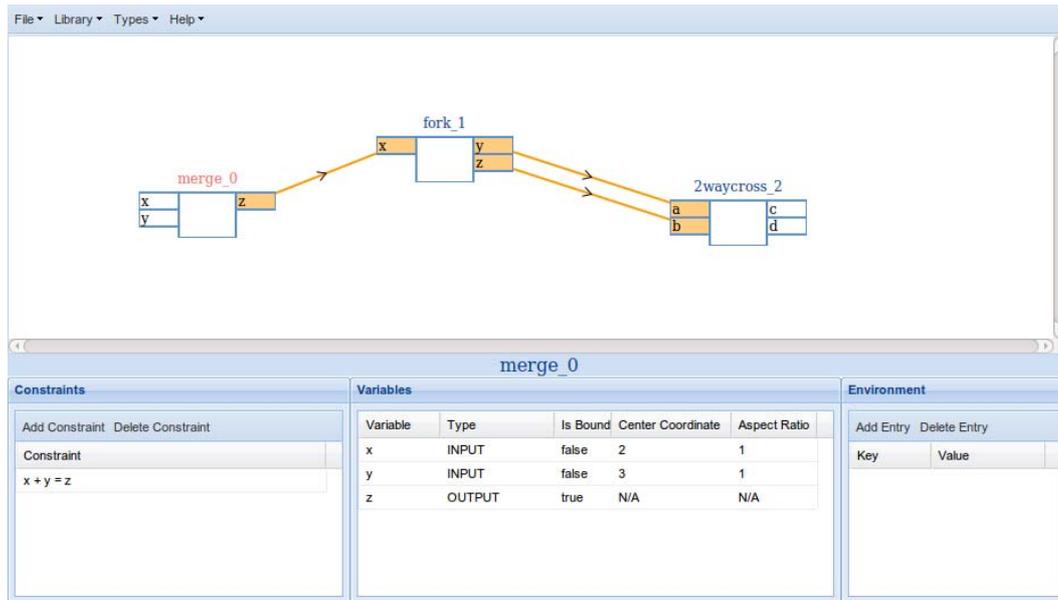


Figure 2. View of a network consisting of 3 connected modules in the NetSketch tool.

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.

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 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 module, and output ports to the right.

⁴In 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; 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 .

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

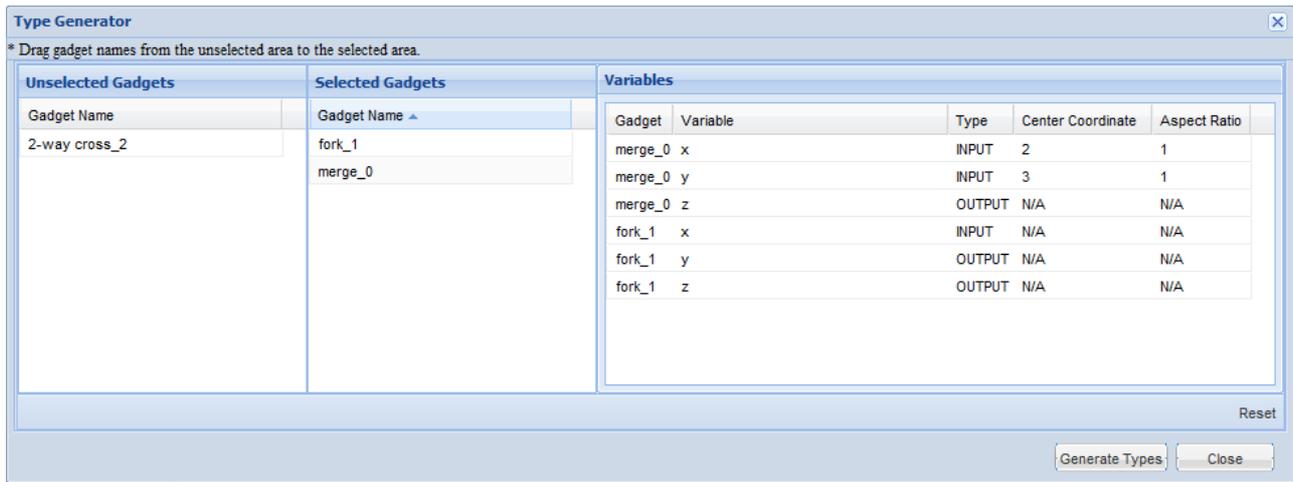


Figure 3. Type generation window with two connected modules selected for type creation.

use than an untyped one, 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.

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.

In its current form the NetSketch tool will provide the generated type to the user, but this information is not retained in the system. In future implementations the tool will transition from *base* mode to *sketch* mode at this point. In *sketch* mode all modules would be typed, and compositional analysis would be performed. 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 6 for more details.

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.

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. These features follow naturally from the NetSketch formalism, and will be added to future implementations.

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.

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

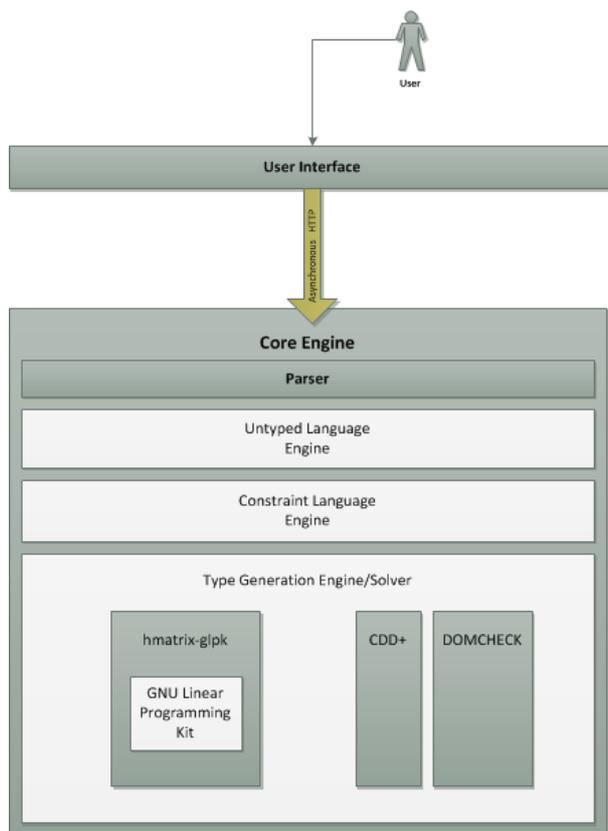


Figure 4. Architecture of NetSketch Tool

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-rectangle⁵ 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 these. 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 pro-

gramming. 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.

The hyper-rectangle can be computed by using linear programming to solve the system of equations and inequalities, first with the objective function $\text{Maximize}(v)$, then again with the objective function $\text{Minimize}(v)$ 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.*, $\forall v \in \mathcal{I}, \text{type}(v) = [\text{Solution}_{\text{Min}}, \text{Solution}_{\text{Max}}]$).

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.

⁵ 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.

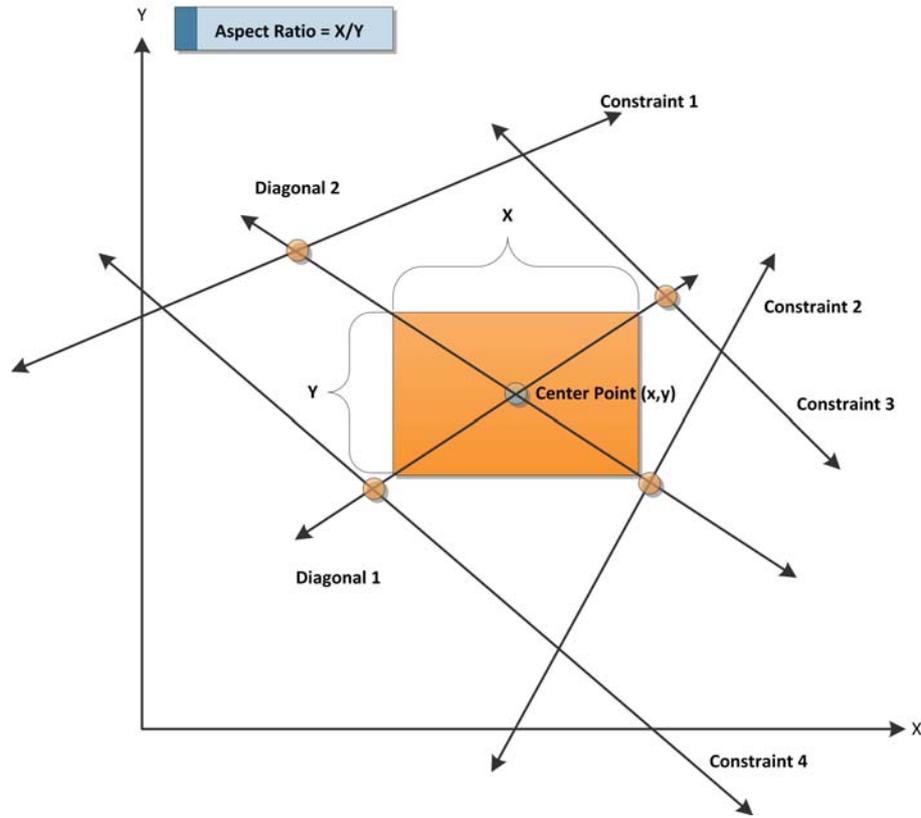


Figure 5. Input Type Generation

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 addition to directly executing functions 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.

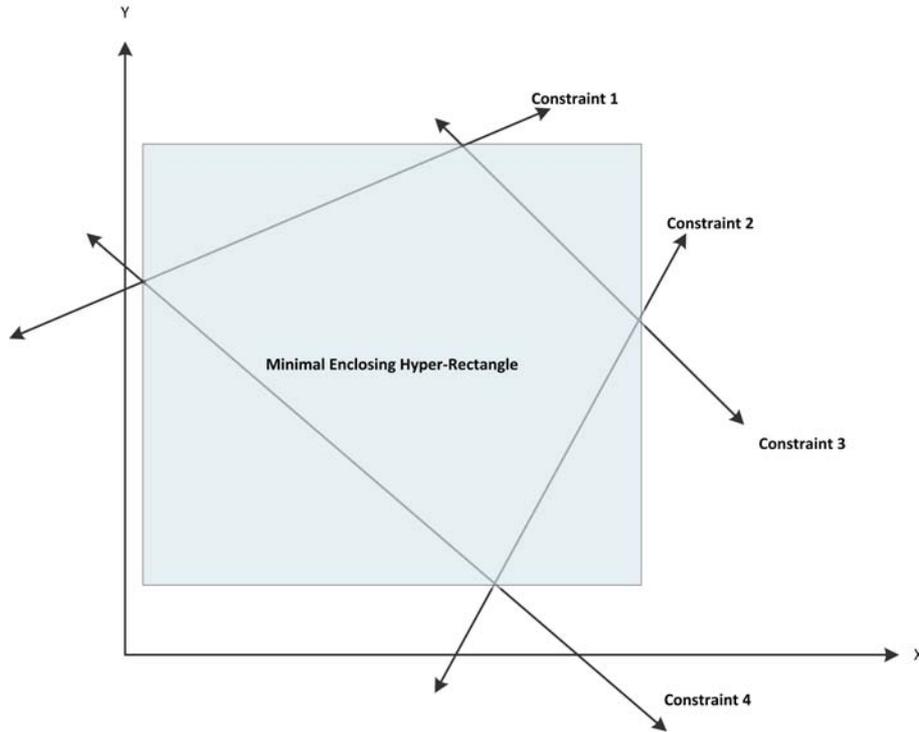


Figure 6. Output Type Generation

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 Appendix A 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.

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 $S_{\text{Min}} \subseteq I \cup O$ where I and O represent the set of inputs and outputs, respectively, of the system.

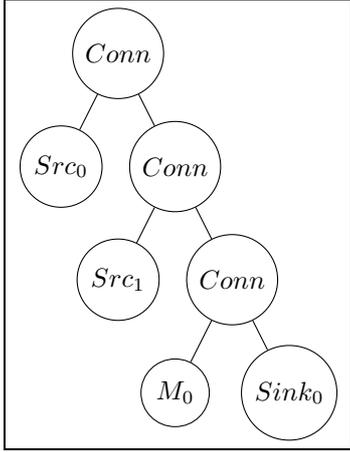


Figure 7. Tree view of the NetSketch network depicted in Figure 8

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.

Appendix A defines two algorithms for constructing \mathcal{S}_{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}_{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 Appendix A 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}_{Min} from which the second can start. This variant allows for significant savings in computation time.

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

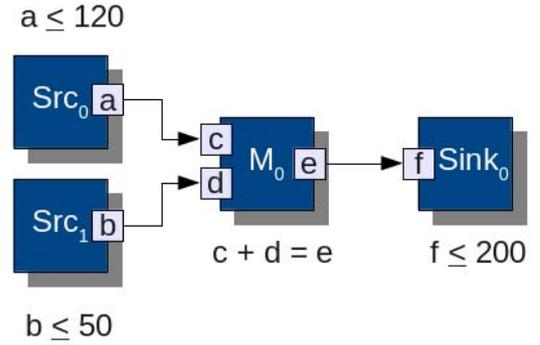


Figure 8. Two source modules, a merge, and a sink.

version thereof) with any equation-based constraints represented directly in the `equation` section of the resulting class definition. For all variables in \mathcal{S}_{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}_{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 provid-

ing 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”.

Tool Completion In its current state the NetSketch tool implements a subset of the features and functionality that are expressed in the NetSketch formalism [4]. The tool will be expanded to allow for a *sketch* mode in which the types generated will be persistently attached to the sets of modules to which they relate. These typed networks can then be analyzed, and various scenarios for satisfaction of safety constraints tested. In addition, the concept of a network *hole* will be introduced into the tool. This idea, defined in detail in [4], allows for the creation of nodes within the network 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 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 implies 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 accessible 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 and detailed in Appendix A 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. Real World Examples of Compositional 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.
- [23] Evgeny Tarasov. HackageDB: hswip-0.3.
<http://hackage.haskell.org/package/hswip>, May 2011.

Appendices

A. Translation In Detail

NetSketch and Modelica overlap to a great extent in their structured representation of models and constraints. Modelica, however, has two facets that limit the ability to do a direct translation from a NetSketch model. First, the constraints governing a Modelica model are restricted to equations [1]. NetSketch constraints are in fact a parameter of the formalism and so are open ended. Even if the constraint space selected is the linear constraint model in use by the current implementation, inequalities are allowed alongside equations. Removing this difference does not alone allow for a direct translation, however, as systems of constraints in Modelica must also be balanced (that is, not over or underdetermined) according to certain rules [1]. Modelica is also largely a tool for simulation with respect to time. This basis contributes to the difficulty of porting NetSketch models, as in its current form NetSketch does not explicitly represent time (though it can be encoded in the type of commodity represented by the flow, and variations of NetSketch have been considered that make time an explicit parameter).

Acknowledging the above difficulties, this appendix outlines an approach that accommodates the differences in the frameworks while allowing for a meaningful translation. In this approach, NetSketch models are analyzed to determine a subset of the system’s variables such that this new set can act as a driver for the entire constrained flow network. Elements of this *minimal covering set* are bound to parameters that can take a single value (*i.e.*, per variable, per simulation). The resulting parameterized model is then transformed into a Modelica model and can be used to test the safety of the system when specific values (or ranges) for the parameters are provided. In this way, specific instances, or multi-dimensional ranges of instances, can be analyzed both for satisfaction of safety constraints and for examination of complete internal state given a partial specification.

A.1 Minimal Covering Set

Due to the inability to include inequality statements directly in the equations section of a Modelica model, an alternative approach must be used. One option is to include the constraints on the right-hand side of an equation as a boolean expression. This approach can be used to ensure validity by introducing a new variable to represent the satisfaction of the constraint set. For example, the constraint $x + y \leq 20$ can be considered by the system using a statement such as: $\text{isValid} = x + y \leq 20$. In this way, the variable isValid will be true when it is the case that $x + y \leq 20$, and false otherwise. We are not aware of a native Modelica mechanism (functionality intrinsic in the simulation-based nature of the system) for solving the set of equations/inequalities in order to determine for what values of x and y this statement holds. Modelica’s coding language is indeed powerful enough to express algorithms

for performing these mathematical calculations, but taking that approach simply uses the procedural aspects of Modelica to replace the Haskell/C code that exists currently for type generation. In this case we do not gain an advantage over the current implementation and have instead simply re-implemented the algorithm in a less mainstream language. What we can do with these new statements, however, is simulate the system under consideration given specific concrete values for x and y (continuing the example above) and ensure that the constraints hold. This would be of only marginal benefit if this binding to concrete values was required for all variables in the system. Instead, we can define a minimum covering set $\mathcal{S}_{\text{Min}} \subseteq I \cup O$ where I and O are the sets of (external and internal) inputs and outputs of a system. \mathcal{S}_{Min} is the smallest set of variables that, when considered along with the other constraints of the system (including the connections between modules), can completely determine the flow if they are bound to concrete values.

As an example, consider Figure 9. Here, six 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 (*e.g.*, 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. In this way, the NetSketch model can be analyzed to determine a minimal covering set \mathcal{S}_{Min} , and can then be transformed into a Modelica model with \mathcal{S}_{Min} as parameters.

A minimal covering set is not necessarily unique. Setting $\mathcal{S}_{\text{Min}} = \{a, f\}$, for example, creates a cover of the same size as does $\{a, b\}$, and will also allow all variables to be determined. The algorithms described in this section find a single minimal cover; however, with simple extensions they could be modified to return all minimal covers, allowing the user to select the most desirable one for their purposes before proceeding with the translation.

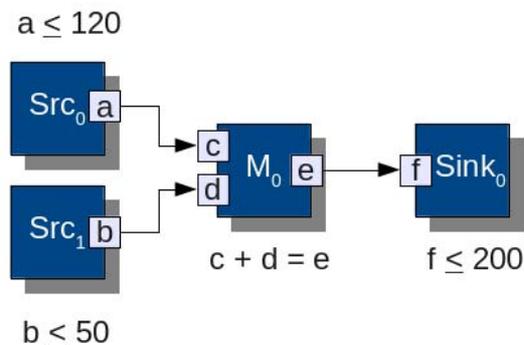


Figure 9. Two source modules, a merge, and a sink.

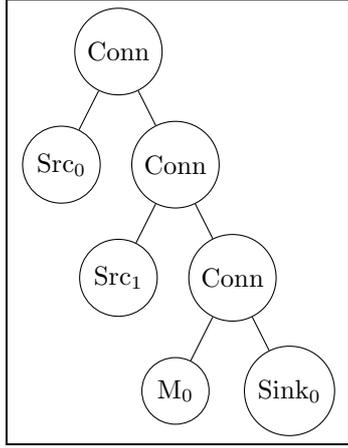


Figure 10. Tree view of the network in Figure 9

To determine the minimal covering set, consider the NetSketch networks described in tree form as in Figure 10.⁶ Two algorithms will be described. The first produces a set that is not always minimal, but is efficient (polynomial) in its runtime. The second is always minimal, but in isolation may run in exponential time. A hybrid approach is also described that uses the first algorithm to produce a baseline from which the second algorithm can start, potentially leading to a large reduction in running time.

A.1.1 Sub-Optimal Efficient Algorithm

This algorithm operates efficiently, as it is based on notions of causality (*i.e.*, flow through input ports may impact values at output ports, but not vice versa), and thus the model may be analyzed in one direction. When translating to an acausal system such as Modelica, however, such an assumption may not be made (*i.e.*, a downstream variable may impact one upstream). Thus, this algorithm will produce a reduced set of variables, but this set may in fact not be minimal, as flow in both directions must be considered. Note that it is not sufficient to simply run this algorithm twice, once for the forward direction and once for the reverse, and then to select the minimum of the two results. Each subgraph of the network may independently benefit most (*i.e.*, the variable set is reduced to the smallest size) from one direction or the other, and thus both directions must be considered simultaneously to achieve a minimum value.

Two passes must be made of the tree representing the NetSketch model. The first pass will build two transition relations R_I and R_E . $R_E(X, x_o, Y, y_i)$ describes transitions between nodes (from output port x_o on node X to input port y_i on node Y). $R_I(X, x_i, x_o)$ will indicate that flow can travel internally through node X from input port x_i to output port x_o via equation constraints. R_I and R_O will be used to assist processing in the second pass. These relations will be constructed by walking the tree and adding new elements to R_E whenever a *Connect* or *Loop* node is encountered, and a new element to R_I whenever a *Module* node is encountered with equation-based (*i.e.*, not inequality-

⁶Non-connected networks (and thus non-tree) may exist in parallel. In this case, these algorithms can be applied to each portion of the network separately.

based) constraints. These constructs define the connections within and between modules and are thus precisely those to be considered when building the transition relations.

The second pass of the tree will actually build the minimal covering set.⁷ The high-level description in Figure 11 defines this algorithm.

A.1.2 Optimal Inefficient Algorithm

To overcome the potentially suboptimal nature of the above algorithm, an alternative is introduced. The algorithm below will find the true minimal covering set(s), and thus is acausal in nature. However, it may have an exponential worst-case running time with respect to the number of variables in the system. A hybrid approach is also described that attempts to take advantage of the speed of the first algorithm and the completeness of the second.

Conceptually, this algorithm first builds a context of propositional statements. Each statement defines whether having a known value for a variable or set of variables necessarily determines the value of another variable. For example, if the output port variable x is connected to the input port variable y then the presence of a binding to a concrete value (directly or indirectly) for x implies that we have an indirect binding to a concrete value for y . The same holds in the opposite direction.

Consider Figure 12. Here, five modules are connected to form a network. Arrows between ports represent NetSketch *Connect* constructs. Since output port a , for example, is connected to input port b , an implicit constraint of $a = b$ exists. Thus, if we have a value for one, we can determine a value for the other, leading to the propositional logic statement $a \leftrightarrow b$. Following this logic across the entire constraint set we get the following base set of statements for Figure 12:

$$a \leftrightarrow b \quad (1)$$

$$b \leftrightarrow c \quad (2)$$

$$c \leftrightarrow g \quad (3)$$

$$d \leftrightarrow e \quad (4)$$

$$f \leftrightarrow h \quad (5)$$

$$g, h \rightarrow i \quad (6)$$

$$g, i \rightarrow h \quad (7)$$

$$h, i \rightarrow g \quad (8)$$

$$\top \rightarrow d \quad (9)$$

Equations that directly relate two variables (including those defined implicitly by *Connect* statements) result in bi-directional implication, as in the first five lines above. Equations relating n variables result in all combinations of $n - 1$ variables implying the single other variable (single direction implication), as in lines (6), (7), and (8) above. If multiple equations relate overlapping sets of variables the $n - 1$ variables on the left side of the implication may be reduced (to $n - 2, n - 3$, etc). In the best case the system is fully determined and thus we do not need to

⁷Note that here the term *minimal* is misused slightly, as the covering set produced, while reduced, may not always be minimal.

- Perform a full search of the tree, maintaining an initially empty set \mathcal{S}_{Min} . For each node perform the following action based on the node-type:

Module

- Examine the constraint set of this module and select those variables v where $v \notin \mathcal{S}_{\text{Min}}$.
- For each variable v consider it's flow type (*i.e.* Input, Output), and it's connection status (*i.e.* Bound, Unbound). If v is:
 - **Input \wedge Unbound:** Add v to \mathcal{S}_{Min}
 - **Input \wedge Bound:** use the relations R_I and R_E from above to determine if it is on a cycle. If so add to \mathcal{S}_{Min} .
 - **Output \wedge Bound:** Construct an initially empty set $\mathcal{C}_{\text{Used}}$. Check if v is related to an input via an equality constraint where that constraint is not in $\mathcal{C}_{\text{Used}}$. If such a relationship exists, add the related constraint to $\mathcal{C}_{\text{Used}}$, else add v to \mathcal{S}_{Min} . In this way each equality constraint can only exclude a single output variable.

Hole

- If the hole is free: add all output and unbound input ports to \mathcal{S}_{Min} .
- If the hole is bound, then a set of constraints exists corresponding to all isomorphisms of the allowed modules (those specified in a Let statement). Use the algorithm for modules (defined above) using a pseudo module representing this hole attached to the mentioned constraint set.

Figure 11. Sub-optimal efficient algorithm

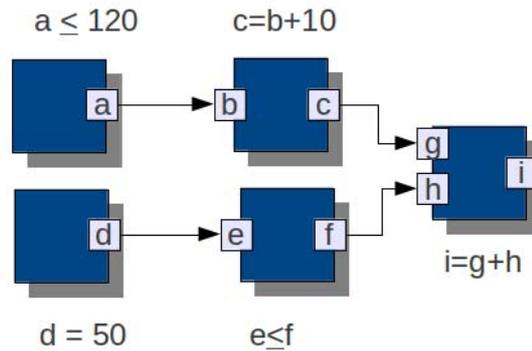


Figure 12. A simple network to be examined via the optimal minimal covering set algorithm

bind any of its variables to concrete values, as the system does so for us. If an equation relates a single variable to a constant we can add a statement such as the one in line (9) above. Inequality constraints do not contribute to the set. The algorithm for producing a minimal covering set is formally stated in Figure 14.

Upon completion of this part of the algorithm a set of propositional statements P will have been created; P will represent a set of rules describing how variable value determination can be conducted. That is, each element in P will describe how knowledge of a value for a variable (or set of variables) implies knowledge of the value of another variable. In addition, P will contain intrinsic truths in the system (*e.g.*, if $x = 50$ is a constraint then a propositional atom representing x will be assumed to be true, as the value of x is known without further binding or inference).

Given P , the problem of producing a minimal cover is now reduced to finding the minimum set of propositional atoms \mathcal{S}_{Min} that must be explicitly assumed to be true in

order to make $\mathcal{S}_{\text{Min}} \wedge P \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$ valid, where C_{Total} is the set of all constraints in the system and Conj is the conjunction of those variables (see Figure 13 for a formal definition). The set \mathcal{S}_{Min} corresponds to variables that, given concrete values, will determine the entire system.

Various algorithms can be defined for finding this minimal set of propositional atoms. Here, three are briefly described:

Linear Search The most straightforward algorithm involves examining all subsets of atoms in the system of a given size starting with 0, and increasing up to the set which includes all atoms. Let $n = |\text{Vars}(C_{\text{Total}})|$. This algorithm simply tests $P \wedge V_i[j] \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$ in a nested loop with i as the loop iteration counter increasing from 0 to n , and j as the inner loop iteration counter increasing from 0 to $\binom{n}{i}$. Here, $V_i[j]$ represents the j th element of the ordered set of all i -combinations of variables from $\text{Vars}(C_{\text{Total}})$ (any ordering of each set is acceptable). For

- **Vars(C)**: a unary function that returns the distinct variable names in a constraint. Formally, given a constraint C of the form $c_0x_0 + c_1x_1 + \dots + c_nx_n \text{ OP } c_{n+1}x_{n+1} + \dots + c_mx_m$ where $OP \in \{\geq, \leq, =\}$, $\text{Vars}(C)$ will return $\{x_0 \dots x_n \dots x_m\}$.
- **Constrs(M)**: a unary function that given a representation of a module as parameter M , returns the set of all constraints contained within M .
- **EqnRel(C, M)**: a binary function that given a constraint represented by parameter C will return a related set, E , consisting of all equation based constraints in $\text{Constrs}(M)$ such that $\forall E_i \in E : \text{Vars}(E_i) \cap \text{Vars}(C) \neq \emptyset \wedge E_i \neq C$. Informally: for each equation based constraint C in a module there is a set of other equation based constraints, E , that does not contain C and whose members have overlapping variables with respect to C .
- **Conj(S)**: a unary function that given a set, S , returns a conjunction of all elements in S . Formally $\text{Conj}(s_0, s_1, \dots, s_n) = s_0 \wedge s_1 \dots \wedge s_n$.

Figure 13. Function Definitions

- Walk the tree representing the network, visiting each node. Maintain an initially empty list of propositional sentences: P , and an initially empty set of metadata about bound Holes: Bound. For each node perform the following action based on the node-type:

Module

- For each equation based constraint C let $E = \text{EqnRel}(C)$ (see Figure 13 for function definitions) and apply this routine:
 - If $|\text{Vars}(C)| = 1$, add the single variable in C , to P .
 - If $|\text{Vars}(C)| = 2$, add $v_1 \leftrightarrow v_2$ to P where v_1, v_2 are the two variables in C .
 - If $|\text{Vars}(C)| = n$ where $n > 2$, then add an implies statement for every $\binom{n}{n-1}$ combination of $n - 1$ variables, where the left hand side consists of the $n - 1$ variables, and the right hand side is the single remaining variable. In addition perform an analysis of the other k constraints defined for that module (where $k \geq 0$) in order to determine if other related constraints allow for stronger implication statements.⁸

Connect or Loop

- Add $v_1 \leftrightarrow v_2$ to V where v_1, v_2 are the two variables being connected.

Let

- Add the pair $(H, \text{Mod}_{\text{Pseudo}})$ to the set Bound. Here H is the hole mentioned in the *Let* statement, and $\text{Mod}_{\text{Pseudo}}$ is a new module constructed by considering the conjunction of all the constraints in the isomorphisms of the allowed modules defined in the *Let* statement.

Hole

- If the hole, H is in the set of the first elements of the pairs in the set Bound then use the algorithm for modules (defined above) applied to the 2nd element of the pair containing H .

Figure 14. Optimal inefficient algorithm

example, given P and a set of variables $\{a, b, c\}$, this would involve testing all the cases in Table 1.

$P \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 0$	$j = 0$
$P \wedge a \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 1$	$j = 0$
$P \wedge b \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 1$	$j = 1$
$P \wedge c \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 1$	$j = 2$
$P \wedge a \wedge b \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 2$	$j = 0$
$P \wedge a \wedge c \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 2$	$j = 1$
$P \wedge b \wedge c \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 2$	$j = 2$
$P \wedge a \wedge b \wedge c \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$	$i = 3$	$j = 0$

Table 1. Cases in linear search for P and $\{a, b, c\}$

In the best case no additional atoms beyond the statements in P would need to be added to the conjunction on the left-hand side. In this case the first test passes and the algorithm stops, producing an empty set \mathcal{S}_{Min} . In the worst case, P is empty (the network contained no equation based constraints, and was comprised of a single module), requiring all three variables a, b , and c to be added to \mathcal{S}_{Min} in order to imply $\text{Conj}(\text{Vars}(C_{\text{Total}}))$.

⁸To do this combine C with each constraint $E_i \in E$ by solving E_i for each overlapping variable of C (generating a set of equations) and for each substituting the resulting expression into C . This forms a set of size > 0 of new equations. Repeat this process recursively by applying it to the result of each element in the new set along with the tail of E . At the completion of this process a set of new equations, all consistent with the original set, will have been generated. Generate implication statements for this set using the procedure described above, though in this pass the expansion of the equation set will not be required. Note that equations

Binary Search A version of binary search can be used to expedite the process of finding a minimal cover. Here, as in the linear search, we perform tests of subsets of conjuncts of atoms from the system along with P , to see if these imply the conjunction of all atoms in the system. The difference is only in the search order. Rather than checking all subsets of size $0, 1, \dots, n$ we select the size of the sets to test according to a binary search. A call to the binary search algorithm passes as a parameter values for the minimum and maximum of the range currently under consideration. Initially the set of sizes to consider is $[0 \dots n]$ where n is defined as in the linear search. We narrow this range through recursive calls to the binary search function. For each test given \min and \max as parameters we find the midpoint m of the range and perform the checks $P \wedge V_m[j] \vdash \text{Conj}(\text{Vars}(C_{\text{Total}}))$ for all values of j , where $0 \leq j \leq \binom{n}{m}$. If none of the checks pass, then we know that no conjunct of atoms of size m in addition to P can imply the set of all atoms, and thus $|\mathcal{S}_{\text{Min}}| > m$. Accordingly we recursively perform this routine on the right side of the range (*i.e.* $[m + 1, \max]$). If, however, one of the tests of size m did pass, then we know that $|\mathcal{S}_{\text{Min}}| \leq m$. Since the true smallest size may be smaller than m we must test the half of the range to the left of the midpoint (*i.e.* $[\min, m]$) via a recursive application of this process. If the range at any given instance of the recursion is represented by $[\text{Range}_{\text{Min}}, \text{Range}_{\text{Max}}]$ then the recursion can stop when $\text{Range}_{\text{Max}} - \text{Range}_{\text{Min}} \leq 2$. In this case, we test the remaining values in the range and select the lowest one as the size of \mathcal{S}_{Min} .

Hybrid Search In the hybrid search we perform either the linear or binary search as above; however, we use the sub-optimal efficient algorithm described previously as a guide. Since the sub-optimal algorithm runs quickly and gives us an answer that is likely to be close to the minimum, we can use the size of this answer as a starting point. For a linear search, we would thus search linearly from 0 to at most the size of the result of the sub-optimal algorithm. For a binary search, we would use the size of the result of the sub-optimal algorithm as the maximum value of the range to initially test, as opposed to using $[0, n]$.

A.1.3 Implementation

Each of these algorithms have been implemented in Haskell. The propositional logic proof engine exists both as pure Haskell code, and as a Haskell interface to SWI Prolog via `hswip`[23]. Incorporating these code sets into the current NetSketch tool will be possible pending the implementation of the actual translation functions as described below.

A.2 Translation

With a minimal covering set defined, a relatively straightforward translation can proceed. The translation again walks the tree representing the NetSketch network. A set

generated further down the recursion stack may make previous equations redundant (*i.e.*, $a \wedge b \rightarrow c$ becomes unnecessary if $a \rightarrow c$ is added to the context). An efficient implementation of this algorithm would detect these and remove the redundant information.

of Modelica classes are constructed from NetSketch modules and holes, and their constraints are represented using a combination of the *equation* section of each class, and a boolean variable present in a single “driver” class. The driver class will be the class that directs the simulation (analogous to a `main()` function in a procedural language). The algorithm is presented in Figure 15 at a high level.

Elements representing Modelica classes, variables, and connections are described using the abstract data types (presented in a Haskell-like syntax) in Table A.2.

The `NET_SKETCH_LIBRARY_ELEMENTS` reference in Table 3 refers to the definition of reusable classes common to all NetSketch \rightarrow Modelica translations. These will include the following:

```
connector OutPort = output Real;
connector InPort  = input  Real;
```

A.3 Simulation

With a Modelica model now available, the system is ready to be simulated. The purpose of the simulation is to determine the safety of the system given a specific set of bindings for the minimal covering set of variables. The user would set these bindings in the initializers of the appropriate module instances in the driver class. The output of importance to the simulation will be the value of the variable `valid`. This variable will be true when the system is safe, and false when the system is not (given the set of bindings). Since the system is not meant to change state over time, the value of `valid` can be examined at any time after time 0 (and thus the simulation need only run for a minimum amount of time).

The above description defines a safety check on a single instance of the model where all parameters (which correspond to the variables in \mathcal{S}_{Min}) are bound. In small models this can be extended to check finite ranges for each parameterized variable in the model. A simple, but inefficient mechanism for this could be achieved by considering the cross product of the ranges of these variables. This would allow for an exhaustive check to be executed by running multiple (parallel or sequential) simulations of the model (one for each element of the cross product). In order for this to be achievable the finite ranges must be converted to finite sets which involves setting a precision level so that a continuous range of real values can be transformed into a discrete set.

Due to the nature of the constraints in the current implementation (linear, conjunctive) gaps of unsafe values can not exist within a range of safe values (*i.e.*, the constraints represent a convex hull). Therefore, a more efficient mechanism exists to check the ranges: it suffices to check combinations of the maximum and minimum’s of all the ranges.

For each variable the ranges of values to simulate must be determined. One option would be to request the user to provide values. In NetSketch terms this corresponds to the users estimating a set of types, and the framework determining if those types are indeed safe. Alternatively, this range selection could be automated; however, calculating

- Perform a full search of the tree maintaining an initially empty set, \mathcal{C} of what will become Modelica classes, along with a single extra Modelica class, d , representing the driver class of the simulation. For each node perform the following action based on the node-type:
 - Loop** Update d to add a new connection to its `connections` attribute. If the child of the loop construct is not a base module, but instead a composite network, that sub-network will need to be examined to find the two (potentially the same) base modules which are actually involved in the connection.
 - Connect** Update d to add a new connection. If either or both of the children of the Connection construct are not base modules, but instead a composite network(s), each non-base sub-network will need to be examined to find the two base modules which are actually involved in the connection.
 - Module** Construct a new ModelicaClass instance, C and add it to \mathcal{C} . Any equation based constraints in the Module will be added to C 's constraints section. Any inequality constraints will be added to the constraints section of d (for later use as part of the boolean expression defining the variable valid). For each variable v in the constraints of the module:
 - If $v \in \mathcal{S}_{\text{Min}}$ then add it to C 's variables section with the `parameter` modifier.
 - If $v \notin \mathcal{S}_{\text{Min}}$ then add it to C 's variables section.
 - If the variable represents an input port then the new variable in C will have type `InPort` (defined above). If the variable represents an output port then the new variable in C will have type `OutPort` (defined above).
 - Hole** Construct a new ModelicaClass instance, C and add it to \mathcal{C} .
 - If the hole is free then for all ports add the variable to C 's variable section, and for those ports that are in \mathcal{S}_{Min} include the `parameter` modifier. If the variable represents an input port then the new variable in C will have type `InPort`. If the variable represents an output port then the new variable in C will have type `OutPort`.
 - If the hole is bound then a constraint set exists corresponding to all isomorphisms of the allowed modules (those specified in the `let` statement). Use the algorithm defined (above) for modules using a pseudo module constructed using the ports of this hole attached to the constraint set mentioned.
- Output a string in the format shown in Table 3.

Figure 15. Translation algorithm

```

ModelicaClass
  modifiers :: [String]
  parentClassNames :: [String]
  variables :: [Variable]
  constraints :: [Constraint]
  connections :: [Connection]

Variable
  modifiers :: [String]
  type :: Type
  name :: String
  initializer :: String

Type = InputPort | OutputPort

Constraint = String

Connection = ((ModelicaClass, Variable), (ModelicaClass, Variable))

```

Table 2. Haskell-like representation of Modelica constructs

the ranges would involve the same process as calculating interval types (as is done in the current system). A hybrid approach could consist of prompting the user for an overly wide range (or allowing the system to guess one), and performing simulations as part of a search on that range to narrow down an appropriate sub-range.

Single simulation time on a dual core AMD II X2 M300 with 4 GB RAM running OpenModelica 1.7.0 on Ubuntu 11.04 takes approximately 3.6 seconds for a small model of 10 variables across 6 modules. Of this time, only approximately 10 milliseconds is used for the actual simulation, with the rest going largely towards compilation time. Since

```

package PACKAGE_NAME

NET_SKETCH_LIBRARY_ELEMENTS

<<Repeated for each class in the set of classes:>>
class CLASS_NAME
  <<list variables in the form:>> [PARAMETER] TYPE NAME
equation
  CONSTRAINTS
end CLASS_NAME

class Driver
  <<List each module represented in the set of driver connections here:>>
  CLASS_NAME CLASS_INSTANCE_NAME [(INITIALIZERS)];
  Boolean valid;
equation
  <<For each connection in the set of driver connections add:>>
  connect(CLASS_INSTANCE_NAME.VARIABLE_NAME, CLASS_INSTANCE_NAME.VARIABLE_NAME)

  valid = if (<<Conjunction of all constraints in driver's constraint set>>)
    then true
    else false;
end Driver
end PACKAGE_NAME

```

Table 3. Output format from translation

the parameters can be updated without recompiling (in OpenModelica via updating [ModelName].init.txt), running many simulations will be quite fast (approximately 327 simulations can occur before simulation time exceeds compilation time, assuming file manipulations to update parameters cost 1 ms). Since each simulation will run with each parameterized variable in \mathcal{S}_{Min} bound to either the minimum or maximum of its range, there are 2^k possible scenarios to test, where $k = |\mathcal{S}_{\text{Min}}|$. This exponential factor quickly limits the practical use of a complete multi-dimensional range test for all but small inputs. The cost of a simulation set is $\text{COMPL} + (\text{SIM} + \text{FILE_UPD}) * (2^k)$ where COMPL is a parameter representing the initial compilation time on a given system for a given model, SIM represents simulation time for that model, and FILE_UPD represents the time to update the parameters file in between each simulation (note the size of the ranges is not a factor in the running time). So on a system where a particular model compiles in 3500 ms, simulates in 10 ms, and file updates cost 1 ms, the equations becomes $3500 + (11 * 2^k)$. This means that a model over 5 variables will take 3852 ms to completely check (assuming all finite ranges), but a model over 20 variables will take approximately 3.2 hours, and one over 100 variables will take more than $4 * 10^{20}$ years. Thus while small models can be checked completely given finite ranges, large models will be restricted to what-if scenarios over single instances.

A.4 Example

Consider the model depicted in Figure 16. This model represents a system consisting of two logically composed components. *Component 1* models two servers producing content, which is then multiplexed onto a single connection and sent through a pass-through node. *Component 2* receives content, passes it through node P_1 , demultiplexes it, and sends it to two receiving clients, Sink₀ and Sink₁. The two components P_0 and P_1 are connected using an “external” connection, as indicated by the dotted line between them.

Within NetSketch this system would be modelled by creating (or re-using from a library) *Component 1* and *Component 2*, possibly created by different modelers. Each component would have a type generated for it. The type would describe the unbound ports at the composite component’s interface. Here, *Component 1* would simply have a single typed output port (corresponding to g in the diagram), with an interval type of $[0, 150]$. *Component 2* would have a single typed input port on its interface (corresponding to h in the diagram), with an interval type of $[0, 125]$.

Since the type T_2 of *Component 2* is not compatible with the type T_1 of *Component 1* (here meaning T_2 is not a surrounding interval of T_1), the NetSketch system would not allow the connection between the two components to be created. We can see that while $[0, 150]$ and $[0, 125]$ do overlap (indicating there are safe values that would allow the systems to be integrated under some scenarios) there

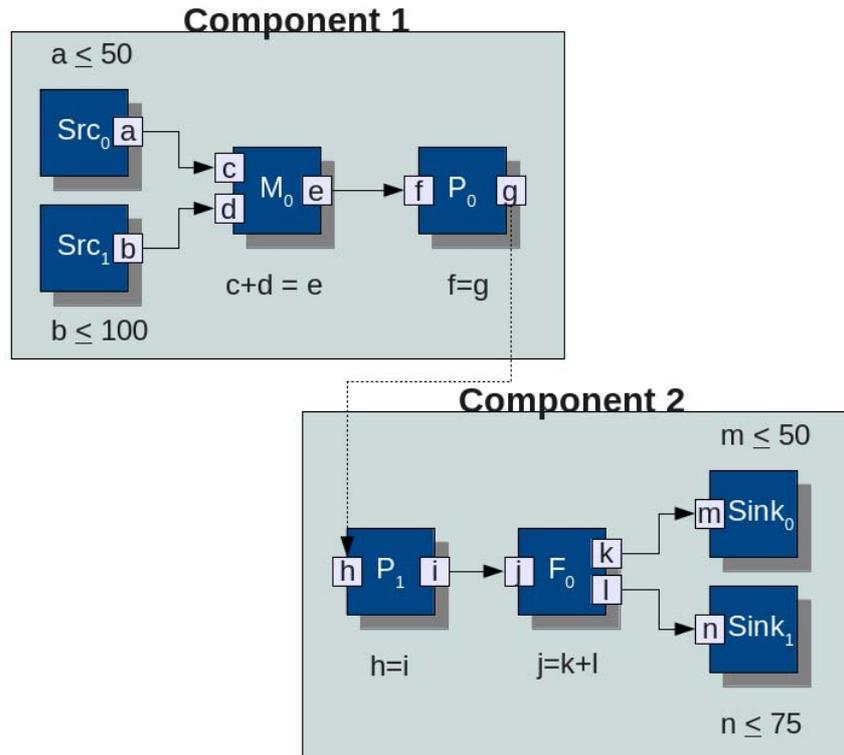


Figure 16. Example Model

are in fact values (namely those in the range $[126, 150]$) that could be produced by *Component 1*, but could not safely be handled by *Component 2*.

Realizing this, a modeler may determine that the composition is unsafe and seek out alternatives, while a designer may modify the network layout or the system internals. For example, a designer may be able to modify the system in such a way that port a on Src_0 may be restricted to output no more than 10 units. In this case a new type is generated for *Component 1* resulting in an interval of $[0, 110]$. Now T_2 is indeed compatible with T_1 since $[0, 125]$ encloses $[0, 110]$, and the two networks can therefore be connected.

A translation to Modelica could be performed on the resulting two-module typed network, however for illustrative purposes we examine this system as a whole (ignoring any structure imposed by grouping modules into the entities *Component 1* and *Component 2*). The minimal covering set, \mathcal{S}_{Min} , will be generated as $\{a, b, l\}$. In this way, the variables in $\{a, b, l\}$ become parameters of the resulting Modelica model, and will be bound to concrete values to allow for simulations. If values are specified for the variables in $\{a, b, l\}$, the entire system will be determined. In this toy example, it is clear which bindings for these parameters will make the system safe, but in larger, more realistic networks in which this is less obvious simulations will help determine exact safe conditions. A single set of bindings may be tested for a “what-if” scenario, or a range of values may be tested to determine the safe boundaries.

Furthermore, exact values of non-bound variables can be examined post-simulation to see the state of the system internals given a particular instance of the model.