2011-12-30

Safe compositional modeling and analysis of constrained flow networks (MA thesis)

BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Thesis

SAFE COMPOSITIONAL MODELING AND ANALYSIS OF CONSTRAINED FLOW NETWORKS

by

NATE SOULE
B.S., Computer Science, Tufts University, 2002

Submitted in partial fulfillment of the
requirements for the degree of
Master of Arts
2012

Approved by

First Reader _____

Assaf Kfoury, Ph.D.
Professor of Computer Science

Second Reader _____

Azer Bestavros, Ph.D.
Professor of Computer Science

ACKNOWLEDGMENTS

A very special thank you to my advisors and mentors Professors Assaf Kfoury and Azer Bestavros. Their ideas, inspiration, encouragement, and friendship have made, and continue to make this journey an exciting and enjoyable one. Their past and present work is the foundation upon which I have built. Thank you also to Andrei Lapets, Vatche Ishakian, and the rest of the iBench team for their help, guidance, and support.

Thank you to my wife Katie, and daughter Maya, whose unlimitted patience and unwaivering support have allowed me to pursue my goals.

SAFE COMPOSITIONAL MODELING AND ANALYSIS OF CONSTRAINED FLOW NETWORKS

NATE SOULE

ABSTRACT

Constrained flow network models represent systems where flows exists between nodes, and constraints exist to regulate those flows. Smart grids, vehicular road travel, computer networks, and cloud-based resource distribution, among other domains all have natural representations in this manner. As these systems grow in size and complexity, traditional analysis and certification of safety invariants becomes increasingly costly. In addition today's techniques require the system to be fully specified in order to perform meaningful analysis. The NetSketch formalism and toolset introduce a lightweight framework for modeling and analysis of constrained flow networks that overcomes these issues. 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. By inferring types for sub-graphs of the modeled networks, not only can cost of analysis be greatly reduced, but 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. This work describes the formalism, a type system, as well as an implementation. In addition potential use cases for this type of modeling and analysis are investigated, and connections are drawn to existing modeling tools and techniques.

Contents

1 Introduction and Overview **1**

1.1 Introduction 1

2 The Formalism **4**

2.1 Overview 4

2.2 Domain Specific Language 5

3 Types **7**

3.1 Overview 7

3.2 Current Type System 8

3.2.1 Input Port Type Inference 9

3.2.2 Output Port Type Inference 11

3.3 Alternate Typings 13

3.3.1 Adaptive Types 14

3.3.2 Unions of Hyperrectangles 20

4 NetSketch Tool **23**

4.1 Overview and Usage 23

4.2 Interface and User Experience 24

4.2.1 Creating Modules 24

4.2.2 Connecting Modules 25

4.2.3 Inferring Types 26

4.2.4 Persistence 27

4.3 Architecture 28

4.3.1 User Interface 29

4.3.2 Core Engine 29

5 Use Cases **32**

5.1 Introduction 32

5.2 Domain: Cloud Service Level Agreements (SLA) 33

5.3 Domain: System Design 41

6 Related Systems **46**

6.1 Harnessing Modelica 47

6.1.1 Modelica as a Computation Platform 47

6.1.2 Translation to Modelica 48

6.1.3 Minimal Covering Set 50

6.1.4 Translation 60

6.1.5 Simulation 62

6.1.6 Example 65

7 Conclusions **67**

Bibliography **71**

List of Tables

6.1 Cases in linear search for P and $\{a, b, c\}$ 58

6.2 Haskell-like representation of Modelica constructs 60

6.3 Output format from translation 61

List of Figures

2.1 Rules for Untyped Network Sketches. 5

3.1 Input Type Generation 9

3.2 Output Type Generation 12

3.3 Two maximally enclosed hyperrectangles of equal size. 14

3.4 A network with no ports bound, followed by that same network with a single port bound. 16

3.5 Union of hyperrectangles approximating the feasible region of a constraint set (from the view of a single variable). 21

3.6 A tree representing the case based type of a network. 22

4.1 View of a network consisting of 3 connected modules in the NetSketch tool. 24

4.2 Type inference window with two connected modules selected for typing. . . 26

4.3 Architecture of NetSketch Tool 28

5.1 Instance specifications 35

5.2 NetSketch model of a cloud configuration depicting both infrastructure provider, and cloud consumer components. 36

5.3 NetSketch model of a cloud configuration for verifying infrastructure meets the requirements of an instance type. 38

5.4 NetSketch model of a cloud deployment for use in inferring the most appropriate instance type. 40

5.5 Instance specifications 41

5.6 NetSketch model of a system of processes communicating via queue based message passing. 43

6.1 Two source modules, a merge, and a sink. 51

6.2 Tree view of the network in Figure 6.1 52

6.3 Sub-optimal efficient algorithm 54

6.4 A simple network to be examined via the optimal minimal covering set algorithm 55
6.5 Function Definitions 56
6.6 Optimal inefficient algorithm 57
6.7 Translation algorithm 62
6.8 Example Model 63

Chapter 1

Introduction and Overview

1.1 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 over those variables 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). Verification of safety invariants across such a system is a critical analysis task, but this task can quickly grow costly as the size and complexity of a model increases. Many systems must also be modeled without the full knowledge of parts of their internals. This may be the case for reasons of privacy (an entity other than the modeler owns/controls part of the system),

timeline (the unknown part of the model is yet to be developed), organizational structure (the model is being created by a large group), among others. Further, various parts of a model may have their constraints best represetned using differing calculi. All three of these issues (cost, unknowns, heterogeneous constraint languages) prove to be challenges for the traditional modeling and analysis methods over constrained flow networks. The NetSketch formalism and accompanying tool offer a constraint-based modeling solution capable of handling such challenges in a light-weight, user friendly maner.

The nodes in a constrained flow network may contain arbitrarily complex constraints that serve to connect its components and regulate its 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 concepts from type theory to simplify the constraints of the network at various levels of the system's composition. A *type* is inferred for 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, a selection of possible type systems, and the current implementation of this formalism. In addition we make connections between this work and the broader constraint-based modeling domain, and begin to investigate potential use cases. In Chapter 2 we describe the domain specific language at the heart

of NetSketch. Next, in Chapter 3 we explore the type system and inference algorithms used in the current implmentation, as well as some alternative designs. In Chapter 4 we describe the NetSketch tool and its architecture. Chapter 5 explores potential use cases for the type of modeling and analysis avaiable through NetSketch. In Chapter 6 we investigate the relation of NetSketch to current constraint-based modeling solutions, examining both how they can be used to provide functionality to NetSketch, and how NetSketch models could fit within their frameworks. Finally, we end with concluding remarks.

Chapter 2

The Formalism

2.1 Overview

In a constrained flow network each node of the 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 sub-system analyses rather than the entire set of constraints. Similar to the benefits provided by modular source code analysis [1] this method allows for efficient analysis of large systems even when the cost of a whole system evaluation might have scaled non-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.

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

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

$$\textsc{Connect}\quad \frac{\Gamma \vdash (\mathcal{M}, I_1, O_1, \mathcal{C}_1) \qquad \Gamma \vdash (\mathcal{N}, I_2, O_2, \mathcal{C}_2)}{\Gamma \vdash (\mathbf{conn}(\theta, \mathcal{M}, \mathcal{N}), I, O, \mathcal{C})}$$

$$\theta \subseteq_{1\text{-}1} O_1 \times I_2,\ I = I_1 \cup (I_2 - \mathsf{range}(\theta)),\ O = (O_1 - \mathsf{domain}(\theta)) \cup O_2,$$
$$\mathcal{C} = \{C_1 \cup C_2 \cup \{\, p = q \,|\, (p, q) \in \theta \,\} \,|\, C_1 \in \mathcal{C}_1, C_2 \in \mathcal{C}_2\}$$

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

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

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

Figure 2.1: Rules for Untyped Network Sketches.

2.2 Domain Specific Language

The NetSketch formalism defines a domain specific language for describing constrained flow networks. In its original form [2] 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 2.1.

Module *Module* defines a new node in the network. This node is atomic *i.e.*, not composed of other nodes. Its definition consists of listing the input and output ports, along with any constraints that exist to regulate traffic through this node via these ports.

Hole A *hole* in a network describes an area whose constraints can not be listed in a flat manner as with a module. This may be the case for one of two reasons. The first is that the hole represents a part of the network 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 then 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. The second usage pattern for a hole occurs when any of a number of interchangeable components should be allowed to fill the hole, and the modeler wishes to ensure satisfaction of safety invariants regardless of which component is actually placed in the hole at any given point (see the *let* construct below).

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 composed with others.

Loop *Loop* allows for the connection of an output port of a network with 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. Any of the given networks, placed in any potential orientation, should allow for the safe operation of the larger system as a whole.[2]

[2]Alternative semantics for the *Let* construct can be imagined (such as "any one of the given networks should allow the system to operate safely"), and have been explored under other names.

Chapter 3

Types

3.1 Overview

In NetSketch, types, akin to those in traditional programming languages, are used to represent approximations of the underlying exact networks. Just as the compiler, via the type system, in a staticly typed programming language can provide certain safety guarantees based on type information alone, so too can the NetSketch processing engine determine safe operations based on its typings. Here instead of applying these types to expressions in a program's text, we apply a type to each port on the interface of a subsystem within the model. These types then act to describe the safe values that can be fed into, or expected out of a particular subsystem. Using this information a sub-typing relation is used to determine safe composition of networks.

The types inferred are approximations, however they are safe approximations in that anything proved about a typed representation of the network is indeed true of the underlying exact network. This system is thus sound, though in most cases will not be complete (*i.e.* there may exist properties that hold in in the underlying exact network, but that can not be proved or are not allowed in the typed version).

In NetSketch a user selects the appropriate subsystem boundaries and a type is then inferred based on the implicit constraints inherent in the topology, and the explicit con-

straints of the selected network. The user may select this boundary based on the existence of logical components within a model, or based on the computational cost of analyzing a network of a given size (of course as the size of the sub-systems selected approximate the size of the entire network, the compositional analysis of NetSketch degrades into whole-system analysis).

The formalism itself does not dictate any one given type system, nor any particular set of allowed constraint languages. Scott in [3] defines a type system as consisting of both a way to define types that can be associated with elements from a language, and a set of rules for type equivalence, compatibility, and inference. Any number of different type systems that fulfill these requirements could be used with the NetSketch formalism to provide the verification properties desired. Below, in Section 3.2 are details of the type system and constraint language used in the current implementation of NetSketch. In Section 3.3 alternative type systems are considered.

3.2 Current Type System

In the current implementation of NetSketch (as described in Chapter 4) the available constraint language allows for linear equations and inequalities. The job of a type is to approximate these constraints in a way that is both as precise as possible, and that is represented in a significantly more efficient and usable form. Here by a more efficient form we mean a representation that allows for quick analysis to determine the network's eligibility for composition with other typed modules, and by usable form it is meant that this form provides an intuitive and helpful summarization of the constraints to the user. To this end the implementation infers open/closed intervals on the real line as the type of a given port.

An input port X with type $[X_{min}, X_{max}]$ indicates that the network in question will be guaranteed to operate safely with respect to its constraints if the range of values coming in via port X is between X_{min} and X_{max}, and all other input ports take in values that

respect their types. An output port Y with type $[Y_{min}, Y_{max}]$ indicates that the network in question will be guaranteed to output via Y values between Y_{min} and Y_{max}.

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

3.2.1 Input Port Type Inference

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 3.1 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.

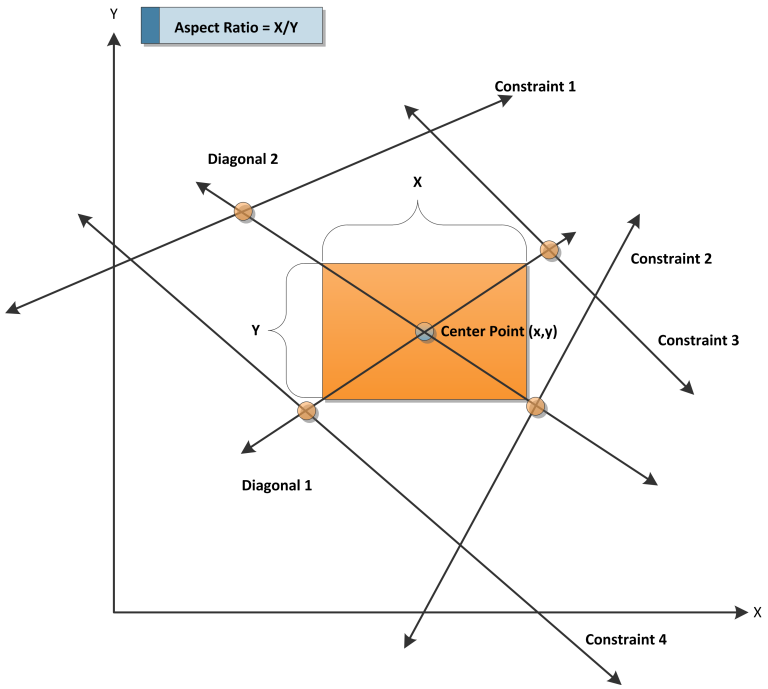

Figure 3.1: Input Type Generation

To create intervals for the input variables we need to find a largest enclosed hyper-rectangle[1] within the convex hull. Each dimension of this hyper-rectangle will then represent one of the input ports, and the boundaries of the hyper-rectangle will become the interval-based type. A maximally enclosed hyper-rectangle is not necessarily unique for any given convex hull. A unique area is required, however, so as to ensure a consistant typing scheme. 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 types is underway as described in Section 3.3. For this implementation, a process was used that involves asking the user to provide a center point for the hyper-rectangle, along with a multi-dimensional aspect ratio relating all input variables. This aspect ratio in effect defines the shape of the hyper-rectangle. In Figure 3.1 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 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 3.1). 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 3.1 are high-

[1]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.

lighted 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, C_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 by the use of the *Connect* or *Loop* constructs (or when in the context of the implementation of NetSketch, in the visual connections drawn between modules and 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.

3.2.2 Output Port Type Inference

As with input type inference, here it is helpful to visualize the linear constraints as forming a convex hull as depicted in Figure 3.2. 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.

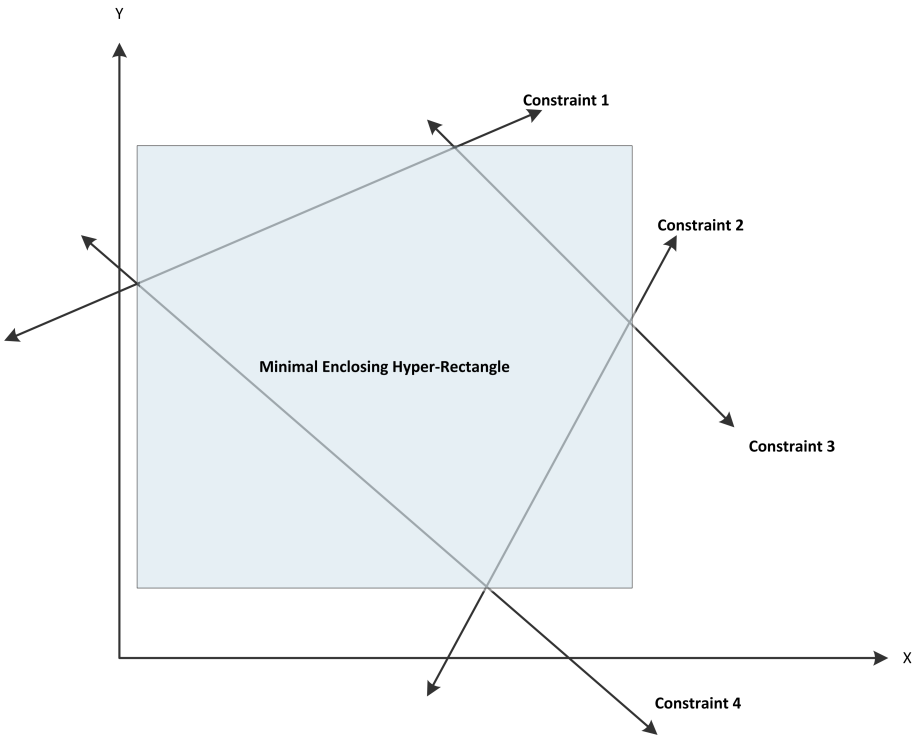

Figure 3.2: Output Type Generation

The hyper-rectangle can be computed by using linear programming to solve the system of equations and inequalities, first with the objective function Maximize(v), then again with the objective function 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. 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.

3.3 Alternate Typings

In the current version of the NetSketch implementation constraints consist of linear equations and inequalities. This results in the constraint sets representing convex hulls. To infer an input type (a set of "safe" values for input to the network) based on an interval typing system, a maximally enclosed axis-aligned hyper-rectangle may be identified. Such a system allows for efficient type checking. It, however, also suffers from two issues:

- Non-unique maximally enclosed axis-aligned hyperrectangles may exist
- Overly conservative approximations of the feasible region

The first item refers to the fact that in any given convex hull there may be more than one maximally enclosed hyperrectangle as depicted in Figure 3.3. Since the system uses intervals as the types for individual ports, the user is required to provide further information in order to select among the possible hyperrectangles. In the current version of the tool the user is prompted for a center point and aspect ratio. Given these two values a unique enclosed axis-aligned hyperrectangle can be identified. The requirement of these extra elements may be viewed positively or negatively depending upon the needs of the user. In some scenarios a type interval centered around a key point known to the user may in fact be more desirable than the global optimal type interval, which may in fact not include the user's preferred region. In this case the use of a user provided center point is indeed desirable. Similarly the user may have some preference between the various ports in regards to which are favored for maximization of their corresponding intervals, or there may exist a natural trade-off between dimensions (3 extra units of dimension x may result in the same benefit as 1 extra unit of dimension y). In these types of cases the multidimensional aspect ratio is also a desirable configuration point. In other scenarios the user may not have a known point/region of interest, or desired relationship between the

dimensions. In these cases unfortunately the user is required to think geometrically about the domain they are modelling. This may involve a paradigm shift which can become unwieldy for dimensions higher than 3. Further, when no known/desired region of interest exists, the center point and aspect ratio provided by the user may in fact lead to non-optimal solutions (the resulting enclosed hyperrectangle may not be maximal for the given convex hull).

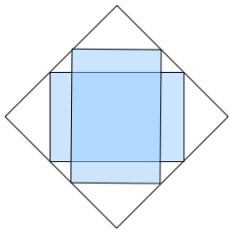

Figure 3.3: Two maximally enclosed hyperrectangles of equal size.

In addition, as alluded to in the 2nd point above, and as can be seen in Figure 3.3, any given hyperrectangle may be quite a conservative approximation of the feasible region, disallowing many "safe" inputs. For example given the constraint set $\{\mathsf{x} + \mathsf{y} \leq 20, \mathsf{x} \geq 0, \mathsf{y} \geq 0\}$ a maximally enclosed hyperrectangle would be defined by the center point (5,5) and the aspect ratio 1:1. This would result in types for x and y of: $\mathsf{x} : [0, 10], \mathsf{y} : [0, 10]$. This disallows the binding of x to an output port of type $[2, 17]$, for example, despite the fact that this is indeed safe given the new condition that y be in the interval $[0, 3]$.

3.3.1 Adaptive Types

What the above example highlights is the potential for the types in NetSketch to be more flexible or the system to provide more guidance by allowing the types to be inferred in a more adaptive manner. Here by adaptive we mean that the exact value of the type itself may change as the state of the system changes (where state in this context refers to the current set of connections for the network in question). The type, dictating the allowed range of values for a given variable, will change as the network that the port is a part

of is restricted or made less restrictive by the act of creating and destroying connections (as described below this benefit may come at the cost of the imposition of an ordering of construction/analysis, and so may need to be considered in a restricted form). In the above example, x has one type before any connections are made to y. Namely x can take on values in the range $[0, 20]$. Likewise given no connection has been made to x, y can take on the range $[0, 20]$. Once one of these ports is connected, however, the type of the remaining unconnected port is impacted. If, for example, x is connected to an output port that guarantees its value to be in the range $[5, 12]$, y will have an updated type of $[0, 8]$. Figure 3.4 depicts this dynamic nature of input port types in an example over a system of 5 constraints. Here we can see that before any connections are made both x and y can range over the full width and full height respectively of the convex hull. Once x is bound, in this case to a port guaranteeing input will be in the range $[20, 27]$, the type for y is updated to show the safe range given that x respects the range $[20, 27]$.

While this approach adds significant flexibility and likely a more precise result, it introduces one critical issue that in its base form is at odds with the underpinnings of NetSketch. Here the order in which a model is consutrcted or analyzed does indeed impact the resulting type, and thus the resulting allowed configurations. NetSketch is a compositional system, which makes a stronger statement than simply being modular. The NetSketch formalism is designed such that the order of analysis and construction do not impact the resulting model - the system and user are free to analyze and construct in any order. The adaptive types described here would violate that premise if left unchanged. In some scenarios it may be acceptable to impose a specific ordering. More often however, this form of typing could be applicable if used simply as an out-of-band tool of exploration for a safe type. After the exploration resulted in a defined hyperrectangle the center point and aspect ratio could be reported to the user, who could then include this in the description of the actual network outside of this "exploratory" mode.

The type of a port in this adaptive typing scheme is then defined as a function that can be used in subsequent connections. Shown below is a type description in Haskell notation:

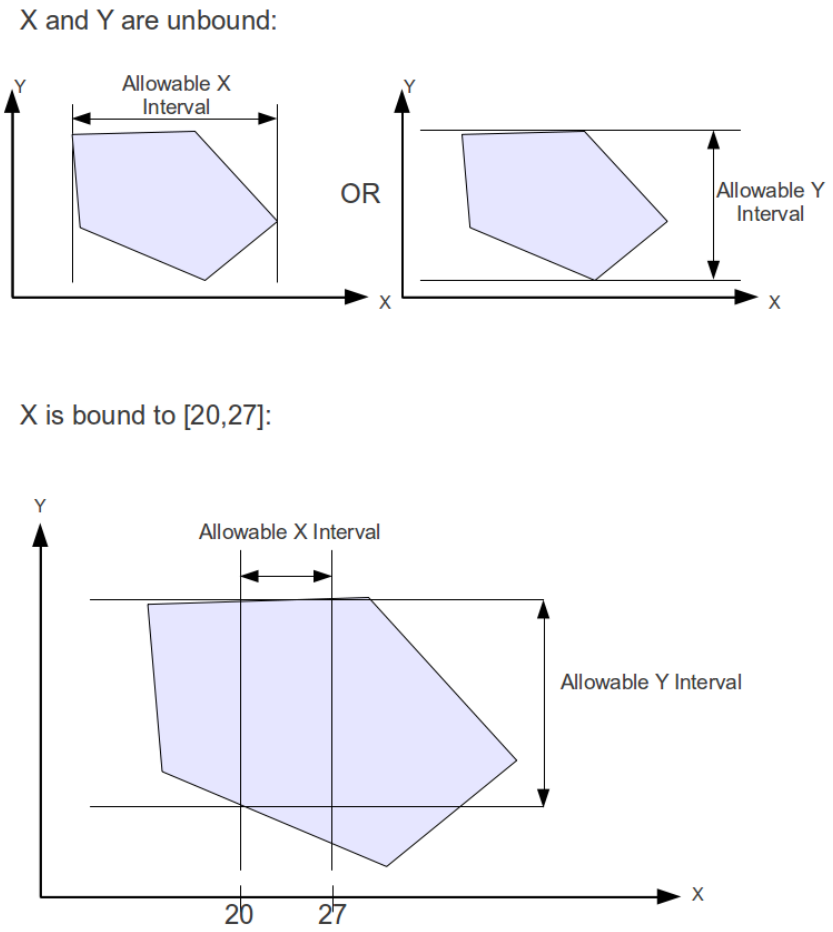

Figure 3.4: A network with no ports bound, followed by that same network with a single port bound.

InputPortTypeFunction :: Interval → Maybe State → Maybe State

Intuitively this says that the type of a given input port is a function given two parameters:

- An interval representing the new restriction being added (*i.e.* the type of the port being connected to)

- The current state of the ports

This function given these parameters will then return an updated state if one is possible. The type Maybe, used here, is borrowed from Haskell to indicate that a State may be returned, but one may not be possible, in which case Nothing is returned.

Here State is defined as $([\text{Variable}], \text{HyperRectangle})$. This represents a pair with the first element as a list of bound variables (for those ports that are already connected), and the second element representing the hyperrectangle of allowed values (for all variables under consideration). Continuing with the example from above, when no connections have yet been made a variable x can have a connection created (and in the process type checked) by applying to it the interval representing the type of the other port in the new connection, and an initial state:

x $[0, 8]$ $([], [x : [0, 20], y : [0, 20])$

Here we are applying two parameters to x. The first is the interval [0,8] indicating we wish to connect x to an output port with type [0,8]. The second represents an initial state. The initial state has the first element of the pair as an empty list indiciating that no connections yet exist, and a hyperrectangle defining the allowed intervals for each variable before any connections have been made. This initial state is inferred using linear programming to find the minimum and maximum of each variable in the system given the constraint set.

The function representing x's type will, in this case, return a new state: $([x], [x : [0, 8], y : [0, 12]])$. This new state indicates that x has been bound, and that the hyperrectangle defining valid ranges now states that x can be between 0 and 8, and y can been between 0 and 12. The state returned from this function call to x will then serve as the current state to any future type checking needed for y. In this way a fully type checked network is the result of a chain of calls beginning with a seed value, *i.e.*

```
fold
   (\currentState → \(var,interval) → var interval currentState)
   initiatState variableBindings
```

We can see that to simply query a port to determine its instantaneous type (*i.e.* the type given the current set of connections), just involves checking the current state. To check the safety of a desired connection just involves application of that connection and the current state to the type representing the port[2].

An alternative approach that simplifies the type signature from one perspective could also be used. In this method the type of a port is:

$$\text{InputPortTypeFunction} :: \text{Interval} \rightarrow \text{Maybe } [(\text{Variable}, \text{InputPortTypeFunction})]$$

Here an initial function generates a list of port type functions with the initial state embeded within them. Calls to a port type function then take an Interval, and return a new list of pairs of a variable name, and a port type function with updated state embedded within. In this way state is never explicitly passed around, but instead new functions representing the port type are generated upon each new connection. Since state is no longer excplicitly passed, checking the instantaneous type of a port without going through the process of updating state is not possible. To accomodate for this the return type of the port type functions could be a list of triples, rather than pairs, with the new element representing the instantaneous type.

Note that as currently described if used alone (i.e. not in the exploratory mode described above, and instead used in a system where an ordering is imposed) these typing schemes provide for less robust scaling than the current design in a system allowing for infinite recursion of typed networks (*i.e.* a group of typed networks can be itself given a

[2]A Haskell based implementation exists for this adaptive type system, though it has not been integrated with the current tool.

type, and that type included in future typings, ad infinitum). This is due to the fact that given a module with n inputs, until that module has connected $\geq n-1$ of those inputs, the original constraint set must be kept (once $n-1$ inputs are connected all ports are bound to exact intervals and the linear constraints can be disregarded for any subsequent typing). In this case if a network contained multiple unbound input ports during each successive typing, the set of linear constraints may not be reduced. This is likely still acceptable in many cases as the size of sub-graphs of the network that become typed networks are likely to be small groupings (and thus more likely to contain small constraint sets) in comparison to the overall network, thus the size of the constraints sets may remain managable. The stronger drawback is instead based on the already mentioned loss of the order-independent compositional nature of the system.

Function Logic

The logic executed as part of the function that embodies the port type operates as follows:

- Add the variable being connected to the list of bound variables
- Generate two sets of constraints, $\mathsf{m}inConstr$ and $\mathsf{m}axConstr$ as:

  $\mathsf{m}inConstr$: for each bound variable (var, (min, max)) generate a constraint var = min

  $\mathsf{m}axConstr$: for each bound variable (var, (min, max)) generate a constraint var = max
- For each variable v in the set of all constraints C of the sub-graph under consideration, use linear programming to find the interval defined by:

  $[\text{Max}(\mathsf{min}\text{Min}, \mathsf{min}\text{Max}), \text{Min}(\mathsf{max}\text{Min}, \mathsf{max}\text{Max})]$ where:

  $\mathsf{m}inMin$ = Minimize v subject to C ++ $\mathsf{m}inConstr$

  $\mathsf{m}inMax$ = Minimize v subject to C ++ $\mathsf{m}axConstr$

  $\mathsf{m}axMin$ = Maximize v subject to C ++ $\mathsf{m}inConstr$

  $\mathsf{m}axMax$ = Maximize v subject to C ++ $\mathsf{m}axConstr$

3.3.2 Unions of Hyperrectangles

An alternative option that allows for adaptive types, but front loads the entire inference computation is to use a union of adjacent (or potentially overlapping) hyperrectangles to approximate the feasible region as seen in Figure 3.5. This approach, similar to the more general adaptive case, is order specific (*i.e.* different types will be inferred based on the order of connections made). This system therefore may find usage as an exploratory/guidance mechanism, but without refinement/extension can not be used as a replacement for the current system without violating the compositional (modular and order independent) nature of NetSketch.

The use of a union of hyperrectangles implies that a connection to an input port for a given variable will match against one or more of the hyperrectangles defined as the input port's type. A hyperrectangle representing the overlapping union of these hyperrectangles will then define what the valid values for the remaining ports are going forward (and this process will continue iteratively for each new port binding). Since this union allows different values for each variable depending upon which of the hyperrectangles a port actually gets bound to, a "view" exists for each variable. A view in this context is a case based inductive structure that provides options for each variable given the state of the variable from which the view is constructed. For example in a two variable scenario, with `x` and `y`, a view would exist from the perspective of `x` which would describe what values `y` could take given various ranges of values for `x`. Those ranges would correspond to the hyperrectangles built, stacked along the `x` dimension. A second view would exist from the perspective of `y`. The number of hyperrectangles to generate for each view would be a parameter of the system, and would dictate how precise the approximation is.

Since this type system is to be adaptive, additional information must be kept after the initial type inference, however, by using unions of hyperrecntables we can keep a tree of intervals representing all views rather than the linear constraints, as depicted in Figure 3.6. Here we see a system of 3 variables, with two paths fully illustrated. The two paths begin

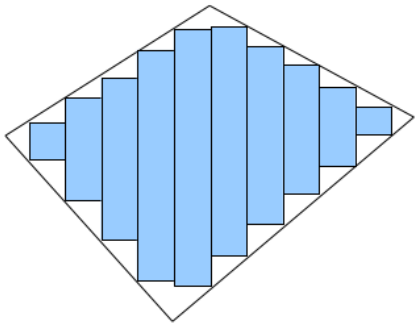

Figure 3.5: Union of hyperrectangles approximating the feasible region of a constraint set (from the view of a single variable).

with variable x, and denote that if x is between 0 and 2 then there are two new views to consider, one from the perspective of variable y, and one from variable z. If after binding the port for x, the user decides to bind y, then we see that if y is between 4 and 8, that z must be between 0 and 2. Alternatively, if after binding the port for x, the user decides to bind z, we see that y must be between 0 and 7. It is possible for the actual desired ranges of a connection to fall within a given case-interval, or across multiple intervals. In the first scenario the tree can be used as is. In the latter scenario the resulting values to use are not those described in the tree, but are derived from those in the tree using the maximum of the minimums given, and the minimum of the maximums given as the bounds to use.

The trade offs to be made when considering the more general adaptive type system described earlier versus this system of unions of hyperrectangles are related to precision, timing, and storage. The system employing the union of hyperrectangles is less precise, in general, than the earlier system described. Since the number of hyperrectangles is configurable, of course this can become more and more precise as this count moves towards infinitity. For partially unbounded constraint sets, however, a union of hyperrectangles must neccessarily be artificially cut off in order to achieve a finite tree representing the views, leading to a less precise representation than the general model. From a timing perspective the two systems each offer different approaches. In the case of the general system, a small amount of non-trivial computation is performed at each point that a port

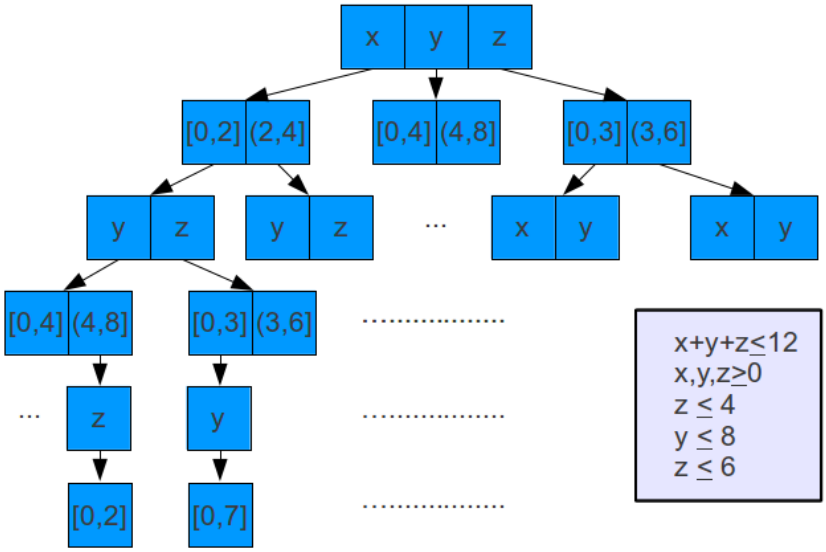

Figure 3.6: A tree representing the case based type of a network.

is connected. In the union of hyperrectangles approach the computation is all front loaded, with a longer initial calculation, but all subsequent bindings work simply from the tree generated. In terms of storage, the union of hyperrectangles can be more compact if there are many constraints over few variables, but for most realistic scenarios will be greatly more verbose as it must keep an exponentially growing tree, as opposed to a structure containing the linear constraints and the current state of the system.

Chapter 4

NetSketch Tool

4.1 Overview and Usage

An implementation of the NetSketch formalism has been created to allow users to express NetSketch concepts via a graphical interface.[1] While the NetSketch domain specific language for constructing networks is simple and intuitive, a graphical environment more closely matches typical users' mental models, and allows for more efficient and error-free modeling than direct use of the textual language. As with any formal analysis system, a low learning curve, and user friendly interface can greatly impact usability and adoption of a system. Thus with NetSketch these principles have been a core of the design from the start. The NetSketch tool offers users the ability to visually create and define modules, add modules from a library of pre-built components, and to create connections among these elements to form *network sketches*. Holes may be defined, and optionaly configured using a GUI interpretation of the *Let* construct. Sub-networks may have types inferred for them, and an infinite recursion of type inference can take place for models of greater and greater abstraction.

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

4.2 Interface and User Experience

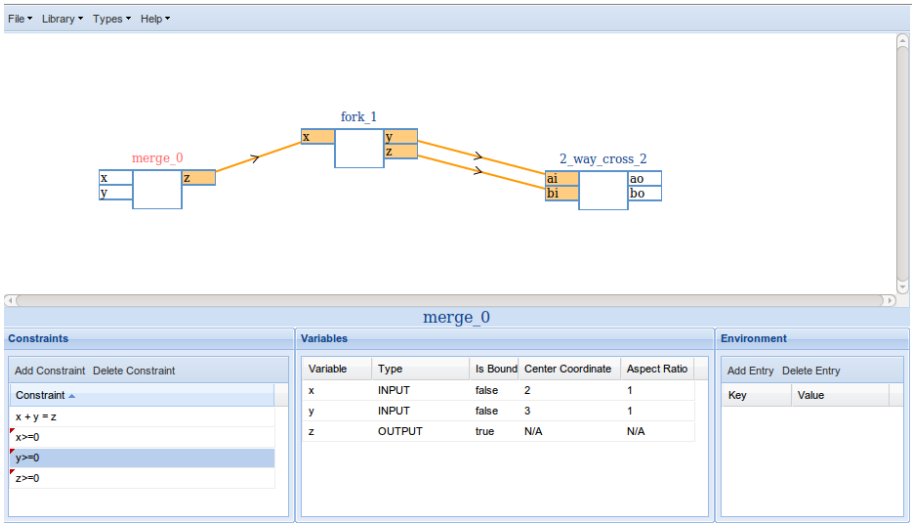

Figure 4.1: View of a network consisting of 3 connected modules in the NetSketch tool.

Figure 4.1 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.

4.2.1 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[2]. 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.

4.2.2 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

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

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.

4.2.3 Inferring Types

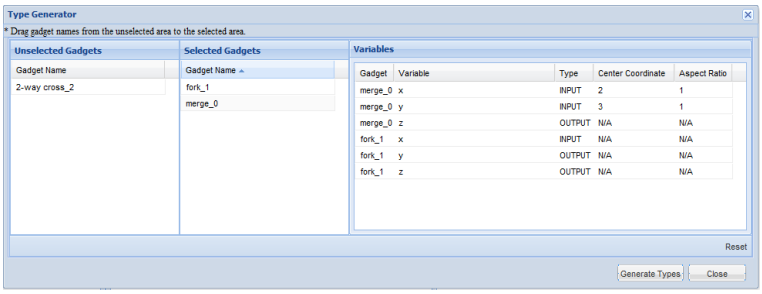

Figure 4.2: Type inference window with two connected modules selected for typing.

When a connected set of modules is in a stable state the user can choose to infer a type for that set. By selecting an option from the menubar a type inference window will open. This window, as shown in Figure 4.2, 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 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. The new container has ports for just the unbound ports on the interface of the underlying exact network that it replaces. 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 inferred 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. Given the current type inference algorithms, 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 3.2 for the reasons behind this requirement and the details of the center-point/aspect-ratio solution, and Section 3.3 for a description of work underway to alleviate this need.

With types inferred, 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.

4.2.4 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.

4.3 Architecture

The NetSketch tool architecture is comprised of a client component and a server component as represented by the **User Interface** and **Core Engine** boxes respectively in Figure 4.3.

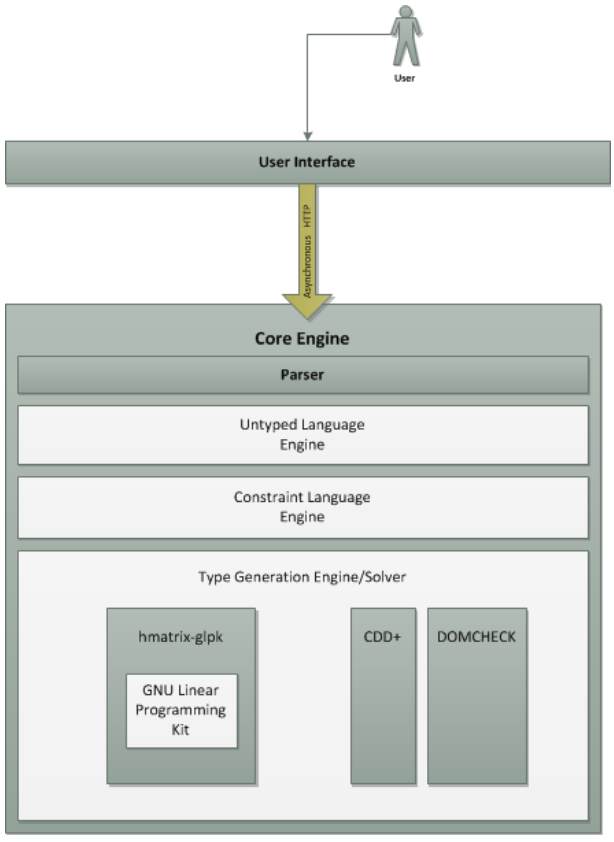

Figure 4.3: Architecture of NetSketch Tool

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.

4.3.1 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 [4] 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) [5], a pure JavaScript vector graphics toolkit, was used. JSGL provided the vector graphics capabilities needed to draw modules, their ports, and the connections between them. JSGL, as with ExtJS, also serves to hide cross browser incompatibilities.

4.3.2 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 [6]. NetSketch uses the built in HTTP server functionality of Happstack to expose the NetSketch API over the web as a form of RESTful web service [7]. 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 [2].

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 BackusNaur Form (BNF). The Haskell parser generator Happy [8] 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+ [9] and Domcheck [10]. 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 [11], around the GNU Linear Programming Kit (GLPK) [12]. 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

second mechanism was developed after creating the HModelica Haskell library (see Section 6.1). In this method NetSketch makes calls via HModelica to an instance of the OpenModelica [13] 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. Given the problem domain, an optimized version of the simplex method, the network simplex method, may be used in many instances. When type inference occurs over a network consisting of linear equations and inequalities where all coefficients are either 0, 1, or -1, this alternative family of algorithms may be used. Here performance can be two orders of magnitude faster [14] than standard simplex method implementations. This dramatic improvement in running time, while important, has not been a critical factor in the current NetSketch implementation, as the size of sub-systems being typed is usually small enough that the standard simplex implementations used provide sufficient performance.

Chapter 5

Use Cases

5.1 Introduction

In order to understand a system, and to best guide its development, it is often critical to understand how the system might be used. He we outline several use cases for the NetSketch formalism and depict them using the current implementation. These descriptions are structured around domains and associated use cases. Each domain provides a brief background, followed by a list of use cases - each of which presents:

- A description of the actor in the use case
- The actor's goal
- A URL pointing to related executable models
- The details of a particular problem instance
- An example of solving the problem instance using NetSketch

Each use case is necessarily (for illustrative purposes) a small example of what would likely be much larger and more complex modeling scenarios. One of NetSketch's core strengths is its ability to scale analyses to large models. These uses cases are intended to show the type, and ease of modeling and analysis in NetSketch, and make evident how these actions would scale along with the model.

Wherever possible the examples use linear constraints to model the problem domain so as to be executable in the current version of the implementation[1]. Where clear benefit from the use of nonlinear constraints was evident, this restriction was abandoned, and is explicitly stated.

5.2 Domain: Cloud Service Level Agreements (SLA)

Background

As cloud computing becomes more pervasive, and more business critical tasks are delegated to the cloud, satisfaction of service level agreements becomes increasingly critical [15–17]. As a prominent player in this space Amazon's cloud offerings must be subject to strict SLAs to provide confidence in their systems. The Amazon Elastic Compute Cloud (EC2) provides compute capacity to end users via virtual machines. Users select from a predefined set of virtual machine instance types depending on their processing needs. Each instance type is bound by a given minimum specification, and associated pricing model. An EC2 standard "small" instance type, for example, guarantees 1.7 GB of memory, a virtual single core processor capable of providing 1 EC2 Compute Unit (ECU), and 160 GB of local storage. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. Note that the Amazon cloud environment is used here to provide concrete examples, however the same prinicples described apply equally well to other cloud based frameworks.

Use Case 1

Actor: Cloud Infrastructure Provider

Goal: Find resource configurations that meet the service level agreement (SLA) requirements of a given user VM instance type.

[1]The NetSketch formalism takes a constraint language as a parameter. The current tool is constructed to work with linear constraints.

URL: http://csr.bu.edu/netsketch/NetSketchClient.html?loadModel=Cloud Producer

Problem Description: A great many configurations of software and phsyical hardware can meet the demands specified in the SLA for any given EC2 instance type. From a technical perspective Amazon need not restrict itself to a particular set of real or virtual components (consistency among components for maintenance and cost drivers is of course a motivating factor in any business model). Thus when designing new configurations to back a given instance type a great deal of lattitude in selection exists. Each potential configuration must be analyzed to ensure that it provides the minimum resources outlined in the instance description. Further, it is in the best interest of Amazon to maximize the financial income from the hardware and software they provide, and thus they must also strive to ensure they are not over provisioning the systems (by either minimizing cost via changing/reducing the provided hardware, or by maximizing profit by increasing the number of virtual instances running on that hardware). Thus relatively tight bounds must be adheared to. This problem can be approached from two opposite directions. A set number of user instances could be selected, and then a configuration searched for that will satisfy these instances. Alternatively a configuration could be established and analyzed to determine the number and type of user instances that could be safely supported by such a configuration.

Example Here we will walk through an example of the problem type defined above. For the sake of brevity this example will be kept to a small size, though the concepts described extend to more elaborate and complex scenarios (indeed one of NetSketch's core strengths is the ability to analyze large systems). In this example a cloud infrastructure provider would like to test if a particular hardware configuration can support three instances of virtual machines: two of type "Small" and one of type "Large". The requirements for these instances are outlined in Figure 5.1[2].

[2]The requirements were selected for the purpose of example from the Amazon instance types listing [18], which may change over time.

Instance Type	CPU (ECU)	RAM (GB)	Storage (GB)
Small	1	1.7	160
Large	4	7.5	850

Figure 5.1: Instance specifications

Here we will concentrate on modeling the CPU requirements. RAM, disk storage, and other resource requirements (*e.g.* network) would be modeled similarly and could be done within the same NetSketch model. Linear constraints are used here, though if real-time processes were to be included in the model, nonlinear constraints may be better suited to the problem. For an example of what form periodic nonlinear constraints might take, see Section 5.3.

To begin we define a phsyical machine on which the virtual machines will run. As we are concentrating on processor requirements we define this machine by its CPU's. Within the NetSketch environment we access the menu `Library` → `Cloud Infrastructure`, and select `Phsyical CPU`. Here we repeat this process 4 times to define a 4-processor machine. In the typical case today these processors are all uniform, however as described in work from the Barrelfish project [19] heterogeneous systems may well provide a more suitable way to meet future processing demands. To demonstrate that homogeneity is not a requirement of the modeling system we will define a heterogeneous system.

The decision of units in Netsketch is left to the modeler. Here, for simplicity, we will work directly with Amazon's Elastic Compute Unit, the ECU. Each processor has a port representing the cycles it produces (in terms of ECU). The variable `out` represents this port. We define CPU_0 and CPU_1 to have $\mathsf{out} = 2$, and CPU_2, and CPU_3 to have $\mathsf{out} = 1$. We introduce a node representing the hypervisor, which will serve to regulate (schedule) which preocesses receive which shares of the total processing power of the physical machine. We select the `Mux-Demux` component from the `Cloud Infrastructure` library. This is a parameterized library element and thus we are prompted to provide the number of inputs, and the number of outputs (with the system building the appropriate constraints based on

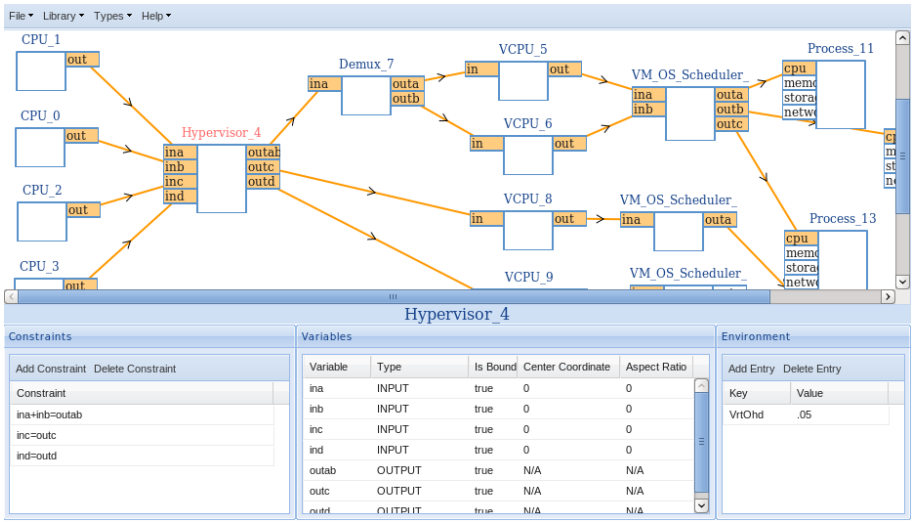

Figure 5.2: NetSketch model of a cloud configuration depicting both infrastructure provider, and cloud consumer components.

our answers)[3]. We have 4 processors that we wish to divide among 3 virtual machines, thus we select 4 and 3 respectively. As depicted in Figure 5.2 we will modify the constraints of this node to capture the notion that two of our virtual processors are pinned to physical processors, while the remaining two virtual processors can float between the remaining physical processors.

The input ports of the Hypervisor node are labelled ina, inb, inc, ind and the output ports outab, outc, outd. We'll route the 2 2-ECU processors to the large instance virtual machine, which requires 4 ECU's, and pin each of the 1-ECU processors to the two small instances. To do this we modify the contraints of Hypervisor to be:

$$\mathsf{ina} + \mathsf{inb} = \mathsf{outab}$$

$$\mathsf{inc} = \mathsf{outc}$$

$$\mathsf{ind} = \mathsf{outd}$$

We desire the output outab to be routed to a single machine, but one with two virtual

[3]The ability to parameterize is not a hardcoded feature of this library element. NetSketch library elements use functions as first class descriptive elements when defining a library model. Thus when the number or relationship of inputs and outputs may vary based on user input this is simply encoded in the function describing the component's inputs, outputs, and/or constraints.

processors. We model this by again using the Mux-Demux parameterized library component to create a 1 input, 2 output node. This is connected to the outab port of the Hypervisor. We then create two virtual CPU's by adding them from the Cloud Consumer library, and connect them to the outputs of the demultiplexer. These library components introduce a new global environment constant, VrtOhd to represent the virtualization overhead cost. The constraints thus regulate the virtual processor to output the cycles it receives from the physical processor, minus a percentage described by VrtOhd. The two processors for the small instance virtual machines are added to the model in a similar way, but connected directly to the hypervisor via outc and outd respectively. In this way they have each been pinned to a single physical processor.

Figure 5.2 depicts the above scenario, with additional model elements representing virtual machine operating system schedulers connected to the virtual processors, and these schedulers connected to particular processes. This system could then be analyzed for satisfaction of the processes' requirements given the physical and virtual structure described. This, however, represents the modeling activities of two separate groups: cloud infrastructre providers (owning the physical resources and virtual infrastructure), and cloud infrastructure consumers (defining their process requirements). A more practical representation is portrayed in Figure 5.3. Here library components representing the specifications of small and large instances are placed on the canvas, replacing the individual processes with a higher level of abstraction. These components contain constraints that require their inputs to meet the minimum requirements of their respective VM instance types, and are thus connected to the output from the 3 virtual machines.

Performing type inference on the entire network will tell us if the system meets its requirements. If no type can be inferred then no range of values on any of the unbound ports can lead to guaranteed safe operation (here safe meaning that the SLA is met). As described the system may not actually be safe. The virtualization overhead represented in the processor constraints on each virtual processor means that the output cycles from these nodes will always be less than the input cycles. Thus while a total of 6 ECU's are

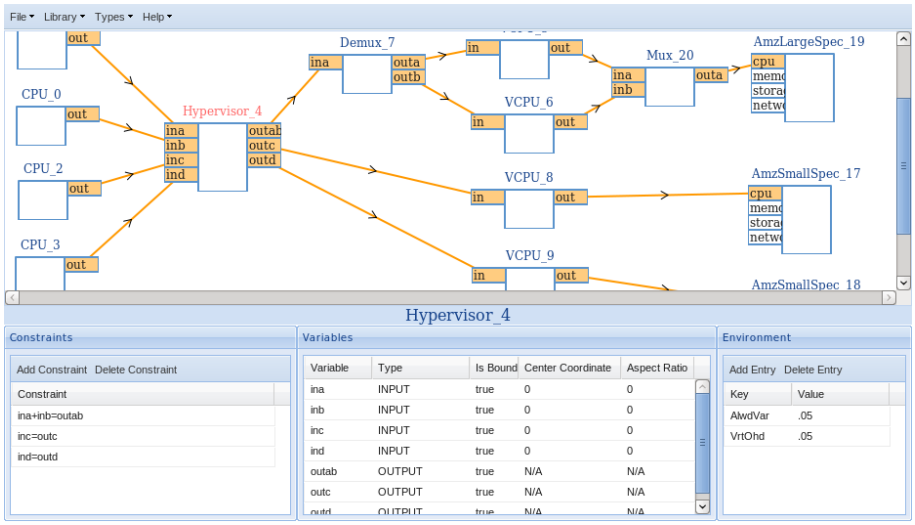

Figure 5.3: NetSketch model of a cloud configuration for verifying infrastructure meets the requirements of an instance type.

generated by the physical processors, and a total of 6 are required, the system leaks some of this processing power to virtualization maintenance tasks. In this case this may actually be acceptable, as an ECU is defined as providing the compute power of a 1.0 - 1.2 Ghz CPU, and thus provides a small amount of allowed variance. In Figure 5.3 this is captured via the AlwdVar environment variable, which has been referenced in each of the instance specification nodes.

Inferring a type for the entire system is not unreasonable in this small example, however doing so essentially reduces the analysis to work on whole-system rather than compositional principles. The true power of NetSketch comes from its ability to use types to allow for greater scalability. As you extend this example to a larger more realistic network, the model in Figure 5.3 would likely be one of many substructures in the entire cloud infrasutrcture. This substructure could be given a type and thus easily reused and composed with other substructures. Formations of disk arrays, network devices/connections, compute nodes, etc. could thus be connected in various ways and recursively typed at various levels of abstraction (cores $\rightarrow$ processors $\rightarrow$ servers $\rightarrow$ racks $\rightarrow$ data centers, etc) and tested to determine how many instances they support, or how well they support a given instance

set. When operating on typed models, even very large networks can be analyzed efficiently, allowing for many what-if scenarios to be tested.

Use Case 2

Actor: Cloud Consumer

Goal: Determine the cheapest instance type that will support application requirements.

URL: http://csr.bu.edu/netsketch/NetSketchClient.html?loadModel=Cloud Consumer

Problem Description: A user desiring to deploy their web application to Amazon's cloud environment must select an instance type from a variety of options. Each option provides different compute capacities, and different pricing. The user wishes to minimize the cost of this cloud deployment while also guaranteeing that their minimum requirements for the web application will be satisifed. The user knows the processes required for their application, and the resource requirments of these processes. They wish to determine the most cost effective, safe, option for their instance selection.

Example Here an example of a model describing cloud consumer instance type selection will be presented. Again here for illustrative purposes only a small model will be described, but the designs documented extended to larger and more complex scenarios.

We will construct a model for a simple web application consisting of 3 core processes/applications:

1. An HTTP server - Apache HTTP Server
2. A Java Web Container - Tomcat
3. A RDBMS - MySQL

As in the previous example we will draw largely from NetSketch's library components, modifying where required. We begin by creating an instance of the `Resource Allocator`

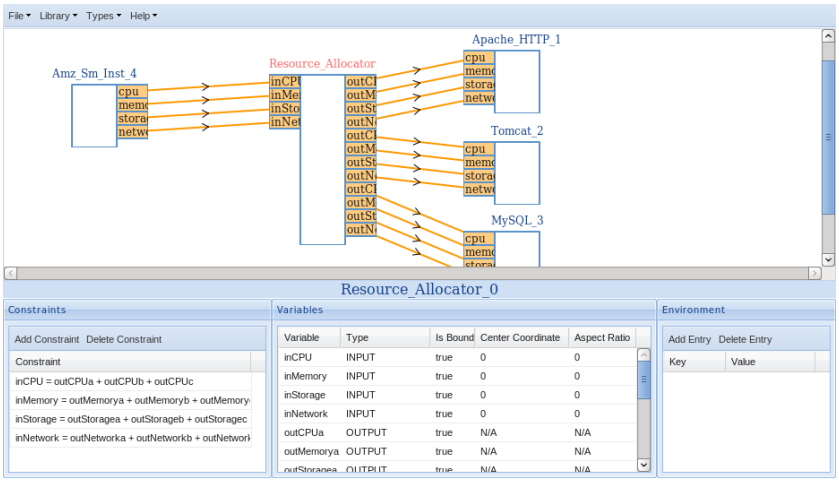

Figure 5.4: NetSketch model of a cloud deployment for use in inferring the most appropriate instance type.

component. This will allow us to model receiving compute capacity, as well as RAM, disk storage, and network bandwidth on the input ports, and scheduling/routing that capacity to 1 or more processes. Since the library cannot predict the number of attached processes to model, the `Resource Allocator` is a parameterized component. Upon selecting it from the NetSketch library we are prompted to provide the number of processes that we wish to connect to this new component. For this example we select 3 in order to represent the processes listed above.

Next we create nodes to represent processes by selecting `Cloud Consumer` → `Process` from the library menu. We do this 3 times, giving each process a descriptive name. The `Resource Allocator` component has 4 outputs for each of the 3 processes representing CPU, RAM, disk storage, and network bandwidth. Connecting these outputs to the corresponding inputs on each process effectively allows routing of each commodity through the allocator.

We now specify the minimum requirements that each of our processes needs in order to operate safely. Here safety may be defined as allowing the processes to execute, or more likely allowing the processes to operate at the speed desired under the load expected (*i.e.*, MySQL may be able to operate given some minimum amount of RAM, CPU, disk, and

network bandwidth, but may need higher levels of some or all of those resources to operate at acceptable performance levels given a particular maximum load).

For this example we specify the minimum requirements as shown in Figure 5.5. No requirements of network bandwidth are listed here as no streaming or intensive throughput use cases are being modeled for this particular application.

Process	**CPU (ECU)**	**RAM (GB)**	**Storage (GB)**	**Network (Mb/s)**
Apache HTTP Server	0.2	205	3.5	0
Tomcat	0.25	185	1.8	0
MySQL	0.4	300	4	0

Figure 5.5: Instance specifications

Figure 5.4 depicts this model connected to an Amazon Small Instance. In this way type inference can be used to determine if the model is safe (*i.e.* the small instance provides enough resource capacity to meet the minimum requirements of the processes). Alternatively the model without the Amazon small instance present could have a type inferred and then this type could be compared to the available instance types to find the best match. A potential extension to the tool could automate this by allowing the user to select an objective function for finding the best match (*i.e.* smallest encompassing bounds, etc) given a set of candidates.

5.3 Domain: System Design

Background

In the design of systems various components are composed and connected to allow data and processing to flow in a desired manner. Each component will produce, consume, or store data - or potentially will exhibit a combination of these behaviors. Examples of such systems can be found in operating system design, enterprise application design, among others. This can be generalized to systems described by queueing network models [20, 21].

Use Case 1

Actor System Designer

Goal Determine required maximum queue depths for a system of processes connected via data flowing through queues.

URL: http://csr.bu.edu/netsketch/NetSketchClient.html?loadModel=Processes And Queues

Problem Description A system architect has designed a series of processes that will interact via passing messages through queues. The maximum rate at which each process will produce messages and the minimum at which each will consume messages is known. A certain amount of resources must be allocated to each queue. The designers desire to limit this resource allocation to the minimum requried to support the expected maximum usage of the system. To do so they must establish the maximum queue depth given the stated rates of consumption and production of messages.

Example In this example a system of 6 processes communicating via 2 queues is modeled and analyzed to determine safe maximum queue depths. To best capture the operation of this system, nonlinear constraints are used. This form of constraints fits well within the NetSketch formalism, but is not executable in the current implementation.

In this model the message production and consumption are periodic. Each process produces/consumes m messages every t milliseconds. m and t are modelled explicitly as outputs or inputs corresponding to production and consumption respectively. A single process could have multiple streams of messages being produced and/or consumed, and thus may have a series of ports $m_0...m_n$ and $t_o...t_n$. Queues have similar input and output ports for allowing messages to flow into, and out of the queue.

In this model we use two library elements, `System Process` and `Queue` from the `Systems` library. Processes and queues are added to the canvas as shown in Figure 5.6. Processes may have simple constraints which explicitly describe their input or output rates.

Proc_1 in Figure 5.6 for example has outa (the number of messages) bound to 100, and outaPeriod (the period) bound to 1000, indicating that this process produces 100 messages every 1000 milliseconds.

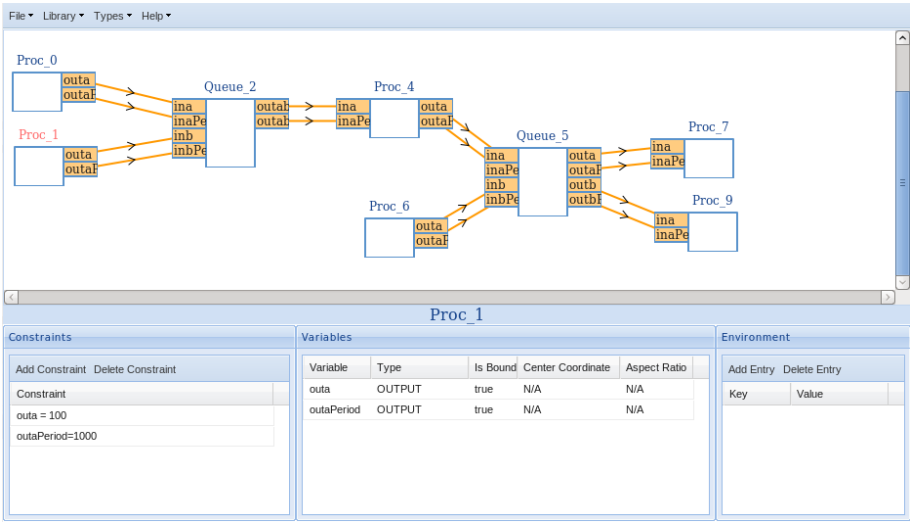

Figure 5.6: NetSketch model of a system of processes communicating via queue based message passing.

While a given queue may accumulate messages over the short term, the system designer wants to ensure that in the long term all messages put on the queue are eventually processed. This property is modeled, for Queue_2 for example as:

$$\frac{\mathsf{ina}}{\mathsf{inaPeriod}} + \frac{\mathsf{inb}}{\mathsf{inbPeriod}} \leq \frac{\mathsf{outab}}{\mathsf{outabPeriod}}$$

The designer may further include a constraint to dictate that the depth of a given queue never goes beyond a certain threshold. Queue_5 for example may contain this constraint:

$$500 \geq$$

$$\mathsf{max}(\mathsf{ina} + \mathsf{inb},$$

$$((\mathsf{floor}(\tfrac{\mathsf{outaPeriod}}{\mathsf{inaPeriod}}) * \mathsf{ina}) + (\mathsf{floor}(\tfrac{\mathsf{outaPeriod}}{\mathsf{inbPeriod}}) * \mathsf{inb})) - ((\mathsf{floor}(\tfrac{\mathsf{outaPeriod}-1}{\mathsf{outaPeriod}}) * \mathsf{outa}) + (\mathsf{floor}(\tfrac{\mathsf{outaPeriod}-1}{\mathsf{outbPeriod}}) * \mathsf{outb})),$$

$$((\mathsf{floor}(\tfrac{\mathsf{outbPeriod}}{\mathsf{inaPeriod}}) * \mathsf{ina}) + (\mathsf{floor}(\tfrac{\mathsf{outbPeriod}}{\mathsf{inbPeriod}}) * \mathsf{inb})) - ((\mathsf{floor}(\tfrac{\mathsf{outbPeriod}-1}{\mathsf{outaPeriod}}) * \mathsf{outa}) + (\mathsf{floor}(\tfrac{\mathsf{outbPeriod}-1}{\mathsf{outbPeriod}}) * \mathsf{outb})))$$

which uses a ternary $\mathsf{max}()$ function to ensure that for the system to be safe it must have a message depth of no more than 500. Alternatively, as the case is in this example, the designer may ask the NetSketch system to determine for them what the maximum safe depth is. One way to accomplish this is by adding a new output port to the queue representing the queue depth. This port, $\mathsf{max}_{\mathsf{depth}}$ would be regulated by a constraint of:

$$\mathsf{max}_{\mathsf{depth}} =$$

$$\mathsf{max}(\mathsf{ina} + \mathsf{inb},$$

$$((\mathsf{floor}(\tfrac{\mathsf{outaPeriod}}{\mathsf{inaPeriod}}) * \mathsf{ina}) + (\mathsf{floor}(\tfrac{\mathsf{outaPeriod}}{\mathsf{inbPeriod}}) * \mathsf{inb})) - ((\mathsf{floor}(\tfrac{\mathsf{outaPeriod}-1}{\mathsf{outaPeriod}}) * \mathsf{outa}) + (\mathsf{floor}(\tfrac{\mathsf{outaPeriod}-1}{\mathsf{outbPeriod}}) * \mathsf{outb})),$$

$$((\mathsf{floor}(\tfrac{\mathsf{outbPeriod}}{\mathsf{inaPeriod}}) * \mathsf{ina}) + (\mathsf{floor}(\tfrac{\mathsf{outbPeriod}}{\mathsf{inbPeriod}}) * \mathsf{inb})) - ((\mathsf{floor}(\tfrac{\mathsf{outbPeriod}-1}{\mathsf{outaPeriod}}) * \mathsf{outa}) + (\mathsf{floor}(\tfrac{\mathsf{outbPeriod}-1}{\mathsf{outbPeriod}}) * \mathsf{outb})))$$

In this way the designer can then infer a type for the system, including this unbound output port, and thus determine the safe range of values for the queue depth. An extension of the current implementation to allow for internal variables would provide an alternative mechanism for defining and inferring the maximum queue depth.

NetSketch's use of compositional analysis techniques brings reasoning about large scale versions of these and other scenarios that may have been formerly intractable into the realm of real world practicality. The NetSketch toolset exposes this power via a lightweight, user friendly mechanism. The use cases presented here illustrate just a sample of the domains

and scenarios that can be naturally represented as equation/inequality based constrained flow networks, and thus modeled and analysed in the NetSketch framework.

Chapter 6

Related Systems

It is important during the development of any system to determine its relationship to the other systems in its context or domain. Well-established constraint-based modeling systems exist today. NetSketch shares a variety of similarities with these tools, but also bears numerous non-trivial differences. The types-based nature of NetSketch sets it apart from many systems in this space. In addition its ability to scale via configurable levels of approximation and abstraction, and analyze systems with under-specified components makes it unique among other related frameworks.

Other notable differences stem from the fact that NetSketch in its current form does not explicitly consider time. Many other constraint-based modeling tools, such as Modelica [22], 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 these time dependent modeling systems, other differences from NetSketch are likely to exist, such as the need for balanced systems and constraints restricted to equations (rather than equations and inequalities) as in Modelica.

Despite such differences, the overlap that does exist offers a great opportunity for various forms of integration. Here, rather than focus on the differences, we focus on

exploiting that area of overlap to the benefit of NetSketch and other systems. In particular we examine two types of relationships: those where NetSketch can consume the services of other systems, and those were NetSketch models can be translated such that they can be analyzed, simulated, or otherwise used in related systems.

In the first category we have examined the functionality of related systems to see if sub-problems handled by NetSketch could be done more simply, efficiently, powerfully, or elegantly by farming out this functionality to existing systems. In this vein we looked at systems like the ECLiPSe constraint logic programming platform [23] for their ability to solve constraints.

In the second category we have examined the relationship between model types of NetSketch and other systems to see how a NetSketch model could be tranformed in a way that would allow it to be executed in the related system.

In both cases we examined a variety of systems (ECLiPSe, Ptolemy II [24], Modelica [22]), but concentrated our efforts on a particular modeling and simulation platform: Modelica.

6.1 Harnessing Modelica

Modelica is an object oriented, equation based modeling language [25]. It is managed by the Modelica Association, though free as well as commercial implementations of the execution environment are developed and maintained by 3rd party organizations.

6.1.1 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 section 6.1.2), 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 [13] implementation to Haskell code. This library, HModelica, enables Haskell developers to create, manipulate, and simulate Modelica models, in addtion 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 number 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 [26]. Here, it will be available for public download and use in the Cabal package format.

6.1.2 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.

The reverse direction, a translation from NetSketch to Modelica, generates models that can be used to perform simulation as a safety analysis tool. 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 [27]. 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 [27]. 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).

Acknowledging the above difficulties, this section 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.

6.1.3 Minimal Covering Set

Due to the inability to include inequality statements directly in the area of a Modelica model that governs the simulation, 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: $\mathsf{isValid} = x + y \leq 20$. In this way, the variable $\mathsf{isValid}$ 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 inference. 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. $\mathcal{S}_{\text{Min}}$ 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 6.1. 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 $\mathcal{S}_{\text{Min}}$, and can then be transformed into a Modelica model with $\mathcal{S}_{\text{Min}}$ as parameters.

A minimal covering set is not necessarily unique. In the above example setting $\mathcal{S}_{\text{Min}} = \{a, f\}$ (among other possibilities), creates a cover of the same size as $\{a, b\}$, 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.

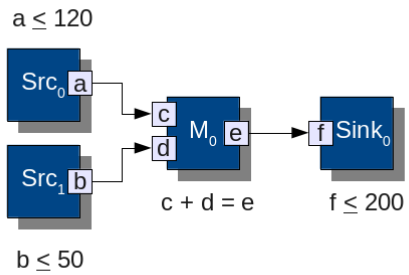

Figure 6.1: Two source modules, a merge, and a sink.

To determine the minimal covering set, consider the NetSketch networks described in tree form as in Figure 6.2.[1] 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

[1]Non-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.

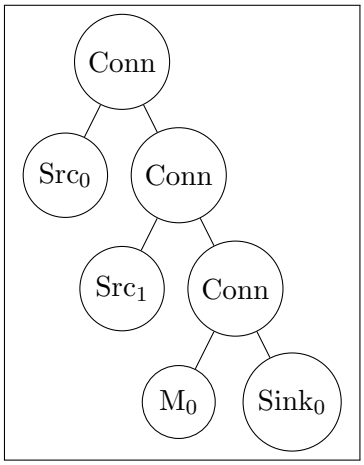

Figure 6.2: Tree view of the network in Figure 6.1

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.

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 indeed restrict 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. $R_E(X, x_\mathrm{o}, Y, y_\mathrm{i})$ describes transitions between nodes (from output port x_o on node X to input port y_i on node Y). $R_I(X, x_\mathrm{i}, x_\mathrm{o})$

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-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.[2] The high-level description in Figure 6.3 defines this algorithm. Since the set of nodes of size N in the tree is scanned twice, with each pass performing constant or linear (with respect to N) operations, this algorithm is $O(\mathsf{N}^2)$.

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 6.4. Here, five modules are connected to form a network. Arrows

[2]Note 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 $\mathcal{S}_{\text{Min}}$. 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 $\mathcal{S}_{\text{Min}}$
    - **Input** $\wedge$ **Bound**: use the relations R_I and R_E from above to determine if it is on a cycle, and no other nodes on the cycle are already in $\mathcal{S}_{\text{Min}}$. If these properties hold then add to $\mathcal{S}_{\text{Min}}$.
    - **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 $\mathcal{S}_{\text{Min}}$. In this way each equality constraint can only exclude a single output variable.

  Hole

  - If the hole is free: add all ouput and unbound input ports to $\mathcal{S}_{\text{Min}}$.
  - 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 6.3: Sub-optimal efficient algorithm

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

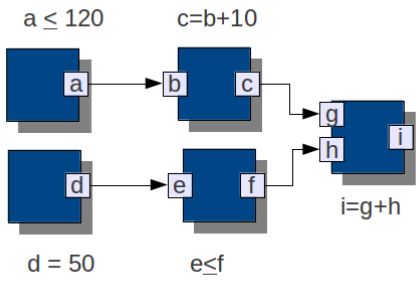

Figure 6.4: A simple network to be examined via the optimal minimal covering set algorithm

base set of statements for Figure 6.4:

$$a \leftrightarrow b \tag{6.1}$$

$$b \leftrightarrow c \tag{6.2}$$

$$c \leftrightarrow g \tag{6.3}$$

$$d \leftrightarrow e \tag{6.4}$$

$$f \leftrightarrow h \tag{6.5}$$

$$g, h \rightarrow i \tag{6.6}$$

$$g, i \rightarrow h \tag{6.7}$$

$$h, i \rightarrow g \tag{6.8}$$

$$\top \rightarrow d \tag{6.9}$$

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 (6.6), (6.7), and (6.8) 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 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 (6.9) above. Inequality constraints do not contribute to the set. The algorithm for producing a minimal covering set is formally stated in Figure 6.6.

- **Vars(C)**: a unary function that returns the distinct variable names in a constraint. Formally, given a constraint C of the form $c_0x_0+c_1x_1+...+c_nx_n$ OP $c_{n+1}xn+1+...+c_mx_m$ where $OP \in \{\geq, \leq, =\}$, Vars(C) will return $\{x_0...x_n...x_m\}$.
- **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 Constrs(M) such that $\forall E_i \in E : \mathsf{Vars}(\mathsf{E}_i) \cap \mathsf{Vars}(C) \neq \oslash \wedge E_i \neq C$. 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 $\mathsf{Conj}(\mathsf{s}_0, \mathsf{s}_1, ...\mathsf{s}_n) = \mathsf{s}_o \wedge \mathsf{s}_1 ... \wedge \mathsf{s}_n$.

Figure 6.5: Function Definitions

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 $\mathcal{S}_{\text{Min}}$ that must be explicitly assumed to be true in order to make $\mathcal{S}_{\text{Min}} \wedge P \vdash \mathsf{Conj}(\mathsf{Vars}(C_{\text{Total}}))$ valid, where C_{Total} is the set of all constraints in the system and Conj is the conjuction of those variables (see Figure 6.5 for a formal definition). The set $\mathcal{S}_{\text{Min}}$ corresponds to a minimal set of variables that, given concrete values, will determine the entire system.

Various algorithms can be defined for finding this minimal set of propositional atoms.

- 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: B*ound*. For each node perform the following action based on the node-type:

  Module

  - For each equation based constraint C let $\mathsf{E} = \mathsf{EqnRel}(\mathsf{C})$ (see Figure 6.5 for function definitions) and apply this routine:
    - If $|\mathsf{Vars}(C)| = 1$, add the single variable in C,to P.
    - If $|\mathsf{Vars}(C)| = 2$, add $\mathsf{v}_1 \leftrightarrow v_2$ to P where $\mathsf{v}_1, \mathsf{v}_2$ are the two variables in C.
    - If $|\mathsf{Vars}(C)| = \mathsf{n}$ where $\mathsf{n} > 2$, then add an implies statement for every $\binom{\mathsf{n}}{n-1}$ combination of $n-1$ variables, where the left hand side consists of the $\mathsf{n}-1$ 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 $\mathsf{k} \geq 0$) in order to determine if other related constraints allow for stronger implication statements.[3]

  Connect or Loop

  - Add $\mathsf{v}_1 \leftrightarrow \mathsf{v}_2$ to V where $\mathsf{v}_1, \mathsf{v}_2$ are the two variables being connected.

  Let

  - Add the pair $(\mathsf{H}, \mathsf{M}od_{Psuedo})$ to the set B*ound*. Here H is the hole mentioned in the *Let* statement, and $\mathsf{M}od_{Pseduo}$ is a new module constructed by considering the conjuction 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 B*ound* then use the algorithm for modules (defined above) applied to the 2nd element of the pair containing H.

Figure 6.6: Optimal inefficient algorithm

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 $n = |\mathsf{Vars}(C_{\text{Total}})|$. This algorithm simply tests $P \wedge V_i[j] \vdash \mathsf{Conj}(\mathsf{Vars}(C_{\text{Total}}))$ 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, $V_i[j]$ repre-

sents the jth element of the ordered set of all i-combinations of variables from $\mathsf{Vars}(C_{\text{Total}})$ (any ordering of each set is acceptable). For example, given P and a set of variables $\{a, b, c\}$, this would involve testing all the cases in Table 6.1.

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

Table 6.1: 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 $\mathcal{S}_{\text{Min}}$. 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 $\mathsf{Conj}(\mathsf{Vars}(C_{\text{Total}}))$.

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 conjucts 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 $0, 1, ...n$ we select the size of the sets to test according to a binary search. A call to the

[3]To 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 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.

binary search algorithm passes parameter values for the minimum and maximum of the range currently under consideration. Initially the set of sizes to consider is $[0...n]$ 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 $P \wedge V_m[j] \vdash \mathsf{Conj}(\mathsf{Vars}(C_{\text{Total}}))$ 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 $[\mathsf{R}ange_{Min}, \mathsf{R}ange_{Max}]$ then the recursion can stop when $\mathsf{R}ange_{Max} - \mathsf{R}ange_{Min} \leq 2$. In this case, we test the remaining values in the range and select the lowest one as the size of $\mathcal{S}_{\text{Min}}$.

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 $[0, n]$.

Implementation

Each of these algorithms have been implemented in Haskell. The propositional logic proof engine exists both as pure Haskell code (emphasizing simplicity, *i.e.* one runtime as prolog is implemented within Haskell), and as a Haskell interface to SWI Prolog via hswip[28]

(emphasizing efficiency via the faster prolog implementation).

6.1.4 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 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 6.7 at a high level.

```
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 6.2: Haskell-like representation of Modelica constructs

Elements representing Modelica classes, variables, and connections are described using the abstract data types (presented in a Haskell-like syntax) in Table 6.1.4.

```
package PACKAGE_NAME

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

<<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 (<<Conjuction of all constraints in driver's constraint set>>)
      then true
      else false;
 end Driver
end PACKAGE_NAME
```

Table 6.3: Output format from translation

- 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 ModelcaClass 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 $\mathcal{S}_{\text{Min}}$ 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 6.3.

Figure 6.7: Translation algorithm

6.1.5 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 $\mathcal{S}_{\text{Min}}$) 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 discretized 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.

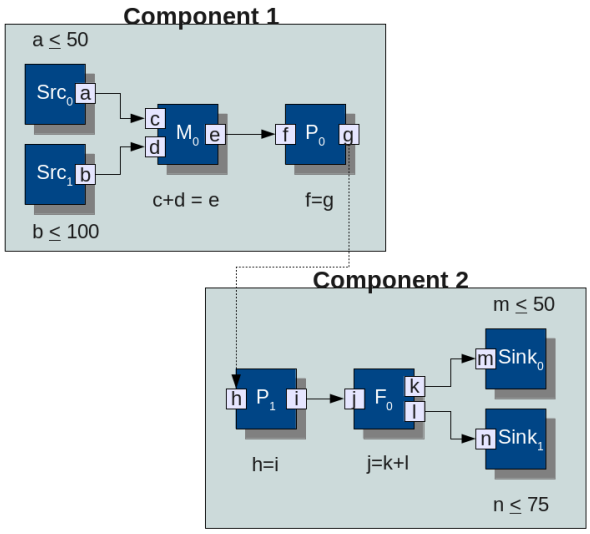

Figure 6.8: Example Model

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 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 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 $\mathcal{S}_{\text{Min}}$ 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 $\mathsf{COMPL} + (\mathsf{SIM} + \mathsf{FILE\_UPD}) * (2^k)$ 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 $\mathsf{FILE\_UPD}$ 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.

6.1.6 Example

Consider the model depicted in Figure 6.8. 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, Sink_0 and Sink_1. 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 inferred 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 an enclosing 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 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 inferred 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, $\mathcal{S}_{\mathrm{Min}}$, 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.

Chapter 7

Conclusions

In this work we have presented NetSketch, a system for modeling and analyzing constrained flow networks. In Chapter 1 we reviewed the problems with traditional whole-system analysis methods, and introduced NetSketch as a potential solution. The inability of whole system analysis to scale as the model in question grow in both size and number/complexity of constraints, limits or slows current modeling efforts of large systems. In addition whole-system analysis requires all modules to have their constraints explicitly expressed, disallowing meaningful analysis over networks containing unknown or under-specified components. We outlined how the compositional nature of Netsketch can address the issue of scalability by allowing for the computationally intensive parts of modeling and analysis to be constrained to small sub-graphs of the network. In addition the compositional properties of Netsketch in combination with its type based approach can alleviate the requirement for fully defining models before analysis can occur. In Chapter 2 we reviewed the theoretical underpinnings based on a new type theory and type system, and discussed the domain specific language for constructing networks. The use of types in NetSketch was compared with that of traditional programming languages and we saw that many of the same principles are at play. Just as how in a staticly typed progamming language the compiler can provide certain safety guarantees based on types alone, in NetSketch the type system can allow us to discard the underlying topology and constraints and focus on

simple types to ensure our safety invariants hold. We saw how the five simple constructs *Module, Connect, Loop, Hole,* and *Let* allow us to build up networks in a compositional manner. The two primary uses of holes were discussed: as placeholders for unknowns in the network, and as placeholders for interchangeable components. We took a more concrete look at the algorithms and methodologies behind the current type system in Chapter 3. Types as open/closed intervals over the real line were presented as approximations of linear constraints. A method for inferring such types based on maximally enclosing and maximally encolsed axis aligned hyperrectangles was presented, before discussing some of its limitations and possible alternatives. In Chapter 4 we looked at the implementation of NetSketch first from the perspective of an end user wishing to make use of the web based toolset, and then from the view of a system architect. The JavaScript, Haskell, and C/C++ components of the tool were discussed, as well as some of the intuition for these particular choices. Chapter 5 explored some potential use cases for NetSketch in the domains of compute resource service level agreements (SLAs), as well as system process communication over queues. Finally in Chapter 6 we examined the relationship of NetSketch to other constraint based modeling tools. These were investigated both as computation platforms, and as a basis for performing translation from NetSketch. The Modelica simulation platform was considered in detail, and algorithms outlining the required steps for translation were presented.

NetSketch presents modelers with the ability to trade off exactness for the possibility to work with systems consisting of large constraint sets that would otherwise be intractable. Through abstraction, encapsulation, and composition NetSketch takes a non-monolithic approach to the problem and in so doing also opens up other benefits such as allowing for a heterogeneous set of calculii to regulate the model, or analysis over under-defined systems to occur. Other formalisms and methods, such as [29], 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 supports the analysis of the model itself.

NetSketch was designed and developed with the desire to be a lightweight mechanism for formal modeling. Despite its basis in a rigorous formalism, every effort was made to hide these details and complexity from the end user. Our belief is that to be of true practical value to a group that may include non-experts, a formal system must not require detailed knowledge of the underlying mathematical principles. Instead the interface should provide a friendly mapping from concepts known to the user to the intricacies of the formalism. These principles embodied in other areas of formal reasoning such as model checkers [30–32], model finders [33], and type systems [34] can be contrasted with the heavier weight approaches common in much of current research on formal methods and the foundations of programming languages (such as the work on automated proof assistants [35], or the work on calculi for distributing computing [36]). Other work on compositional approaches to modeling based on interactions at the interface of modules has been explored via *interface theories* [37], but differs from this work in its basis in automata as opposed to type theoretic notions. We believe the ease of use of the current NetSketch implementation, coupled with the power it exposes via both its automatic verification and through the ability for human interpretation of the meaning of the resulting types - highlights the potential for this type of lightweight approach to formal analysis.

Many areas of continued work related to NetSketch exist. These include refinements, enhancements, and extensions to the formalism itself, to the toolset and its underlying algorithms, and to the relationship between NetSketch and other systems. An investigation of the potential constraint languages and their solution algorithms (as expressed in surveys of constraint solving such as [38]), as well as possible type systems that could be applied to NetSketch - could reveal far more expresive model definition capabilities. Alternate or extended semantics for elements of the DSL, such as additional forms of the *Let* construct to allow for one-must-satisfy vs all-must satisfy interpretations, could likewise allow for more powerful expressions within the formalism. The toolset could be extended to allow even more natural representation of the underlying formalism, and to include further ease-of-use features. Direct translation to Modelica and other modeling systems from within

the tool could allow for hybrid approaches to analysis where both NetSketch style verification as well as simulation based approaches could lead to even stronger guarantees of safety. In addition, further investigation and elaboration on uses cases could serve to help direct future efforts and allow for a better understanding of the system's natural patterns. This type of extension can also lead to a continued build-out of the NetSketch library of components and networks.

Bibliography

[1] D. Jackson and M. Rinard, "Software analysis: a roadmap", in *Proceedings of the Conference on The Future of Software Engineering* (ACM, New York, NY, USA, 2000), ICSE '00, pp. 133–145, ISBN 1-58113-253-0, URL `http://doi.acm.org/10.1145/336512.336545`.

[2] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean, "Safe Compositional Network Sketches: Formalism", Tech. Rep., Department of Computer Science, Boston University, Boston, MA, USA (2009), tech. Rep. BUCS-TR-2009-029, October 1, 2009.

[3] M. L. Scott, *Programming Language Pragmatics* (Morgan Kaufmann, 1999).

[4] Sencha, "Sencha - Ext JS - Client-side Javascript Framework", http://www.sencha.com/products/js/ (2011).

[5] T. Rehorek, "JavaScript Graphics Library (JSGL) official homepage", http://www.jsgl.org/doku.php (2011).

[6] M. Elder and J. Shaw, "Happstack - A Haskell Web Framework", http://happstack.com/index.html (2011).

[7] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big"' web services: making the right architectural decision", in *Proceeding of the 17th international conference on World Wide Web* (ACM, New York, NY, USA, 2008), WWW '08, pp. 805–814, ISBN 978-1-60558-085-2, URL `http://doi.acm.org/10.1145/1367497.1367606`.

[8] A. Gill and S. Marlow, "Happy - The Parser Generator for Haskell", http://www.haskell.org/happy/ (2011).

[9] K. Fukuda, "cdd and cddplus homepage", http://www.ifor.math.ethz.ch/∼fukuda/cdd_home/cdd.html (2011), swiss Federal Institute of Technology.

[10] F. Margot, "Francois Margot Homepage", http://wpweb2.tepper.cmu.edu/fmargot/ (2011), carnegie Mellon.

[11] A. Ruiz, "HackageDB: hmatrix-glpk-0.2.1", http://hackage.haskell.org/package/hmatrix-glpk (2011).

[12] G. P. Developers, "GLPK GNU Project", http://www.gnu.org/software/glpk/ (2011).

[13] O. S. M. C. (OSMC), "Welcome to OpenModelica", http://www.openmodelica.org/ (2011).

[14] J. W. Chinneck, *Practical optimization: a gentle introduction* (Electronic Document, 2004).

[15] P. Patel, A. Ranabahu, and A. Sheth, "Service level agreement in cloud computing", in *Cloud Workshops at OOPSLA09, 2009* (2009), URL `http://knoesis.wright.edu/aboutus/visitors/summer2009/PatelReport.pdf`.

[16] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities", in *The 10th International Conference on High Performance and Communications(HPCC 08)* (2008), URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5767932`.

[17] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis, "Flexible use of cloud resources through profit maximization and price discrimination", in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on Data*

Engineering (2011), pp. 75–86, ISBN 978-1-4244-8959-6, URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5767932`.

[18] Amazon, "Amazon EC2 instance types", http://aws.amazon.com/ec2/instance-types/ (2011).

[19] A. Schpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the barrelfish manycore operating system", in *Proceedings of the Workshop on Managed Many-Core Systems* (2008).

[20] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models* (Prentice-Hall, Inc., 1984).

[21] S. Balsamo, V. D. N. Person, and P. Inverardi, "A review on queueing network models with finite capacity queues for software architectures performance prediction", Performance Evaluation **51**, 269 (2003), ISSN 0166-5316, Queueing Networks with Blocking, URL `http://www.sciencedirect.com/science/article/pii/S0166531602000998`.

[22] M. Association, "Modelica and the Modelica Association", https://www.modelica.org/ (2011).

[23] A. Niederlinski, *A Quick and Gentle Guide to Constraint Logic Programming with ECLiPSe* (pkjs.com.pl, 2011).

[24] U. of California at Berkeley, "Ptolemy ii", http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm (2011).

[25] H. Elmqvist., S. Mattsson., and M. Otter, "Object-oriented and hybrid modeling in modelica", Journal Europen des systmes automatiss **35**, 1 (2001).

[26] H. Community, "Hackagedb", http://hackage.haskell.org (2011).

[27] M. Association, "Modelica Language Specification 3.2", Tech. Rep., Modelica Association (2010), http://www.modelica.org/documents/ModelicaSpec32.pdf.

[28] E. Tarasov, "HackageDB: hswip-0.3", http://hackage.haskell.org/package/hswip (2011).

[29] H. 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* (2008).

[30] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems* (Cambridge University Press, 2004).

[31] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "A nusmv: A new symbolic model checker", International Journal on Software Tools for Technology Transfer (STTT) **2** (2000).

[32] G. J. Holzmann, "The model checker SPIN", IEEE Transactions On Software Engineering **23** (1997).

[33] D. Jackson, "Alloy: A lightweight object modelling notation", ACM Transactions on Software Engineering and Methodology (TOSEM'02) **11**, 256 (2002).

[34] B. C. Pierce, *Types and Programming Languages* (MIT Press, 2002).

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

[36] G. Boudol, "The λ-calculus in direct style", in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (ACM, New York, NY, USA, 1997), POPL '97, pp. 228–242, ISBN 0-89791-853-3, URL `http://doi.acm.org/10.1145/263699.263726`.

[37] L. de Alfaro and T. A. Henzinger, "Interface automata", in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering* (ACM, New York, NY, USA, 2001), ESEC/FSE-9, pp. 109–120, ISBN 1-58113-390-1, URL `http://doi.acm.org/10.1145/503209.503226`.

[38] A. Neumaier, "Complete search in continuous global optimization and constraint satisfaction", Acta Numerica pp. 271–369 (2004), Cambridge University Press.