Safe Compositional Network Sketches: Formal Framework*

Azer Bestavros
Computer Science Dept
Boston University
Boston, MA 02215
best@cs.bu.edu

Assaf Kfoury
Computer Science Dept
Boston University
Boston, MA 02215
kfoury@cs.bu.edu

Andrei Lapets
Computer Science Dept
Boston University
Boston, MA 02215
lapets@cs.bu.edu

Michael J. Ocean
Computer Science Dept
Endicott College
Beverly, MA 09195
mocean@endicott.edu

ABSTRACT

NetSketch is a tool for the specification of constrained-flow applications and the certification of desirable safety properties imposed thereon. NetSketch assists system integrators in two types of activities: modeling and design. As a modeling tool, it enables the abstraction of an existing system while retaining sufficient information about it to carry out future analysis of safety properties. As a design tool, NetSketch enables the exploration of alternative safe designs as well as the identification of minimal requirements for outsourced subsystems. NetSketch embodies a lightweight formal verification philosophy, whereby the power (but not the heavy machinery) of a rigorous formalism is made accessible to users via a friendly interface. NetSketch does so by exposing tradeoffs between exactness of analysis and scalability, and by combining traditional whole-system analysis with a more flexible compositional analysis. The compositional analysis is based on a strongly-typed Domain-Specific Language (DSL) for describing and reasoning about constrained-flow networks at various levels of sketchiness along with invariants that need to be enforced thereupon. In this paper, we define the formal system underlying the operation of NetSketch, in particular the DSL behind NetSketch's user-interface when used in "sketch mode", and prove its soundness relative to appropriately-defined notions of validity. In a companion paper [7], we overview NetSketch, highlight its salient features, and illustrate how it could be used in applications that include: the management/shaping of traffic flows in a vehicular network (as a proxy for cyber-physical systems (CPS) applications) and a streaming media network (as a proxy for Internet applications).

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

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.6 [**Software Engineering**]: Programming Environments; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Evolutionary prototyping*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

General Terms

Languages, Verification, Reliability

1. INTRODUCTION

Constrained-Flow Networks: Many large-scale, safety-critical systems can be viewed as interconnections of subsystems, or modules, each of which is a producer, consumer, or regulator of *flows*. These flows are characterized by a set of variables and a set of constraints thereof, reflecting *inherent* or *assumed* properties or rules governing how the modules operate (and what constitutes safe operation). Our notion of flow encompasses streams of physical entities (*e.g.*, vehicles on a road, fluid in a pipe), data objects (*e.g.*, sensor network packets or video frames), or consumable resources (*e.g.*, electric energy or compute cycles).

Traditionally, the design and implementation of such *constrained-flow networks* follow a bottom-up approach, enabling system designers and builders to certify (assert and assess) desirable safety invariants of the system as a whole. While justifiable in some instances, this vertical approach does not lend itself well to current practices in the assembly of complex, large-scale systems – namely, the integration of various subsystems into a whole by "system integrators" who may not possess the requisite expertise or knowledge of the internals of the subsystems on which they rely. This can be viewed as an alternative *horizontal* approach, and it has significant merits with respect to scalability and modularity. However, it also poses significant challenges with respect to aspects of trustworthiness – namely, certifying that the system as a whole will satisfy specific safety invariants.

The NetSketch Tool: In recognition of this challenge, we have developed NetSketch – a tool that assists system integrators in two types of activities: modeling and design.

As a modeling tool, NetSketch enables the abstraction of an existing (flow network) system while retaining sufficient information about it to carry out future analysis of safety properties. The level of abstraction, or sketchiness (and hence the amount of information to be retained) is the result of two different processes that NetSketch offers to users. The first process is the identification of boundaries of the subsystems to be sketched. At the extreme of finest granurality, these boundaries are precisely those of the interconnected modules that make up the system – *i.e.*, the constituent subsystems are the modules. At the other extreme, these boundaries would enclose the entire system. The second process is the control of the level of precision of information retained for the specification of a given subsystem, which are expressed as constraints defined over flow variables at the boundaries of that subsystem. By making conservative assumptions (*e.g.*, restricting the set of permissible inputs to a subsystem or extending the set of possible outputs from a subsystem), it is possible to reduce the complexity of these constraints.

As a design tool, NetSketch enables the exploration of alternative safe designs as well as the identification of minimal requirements for missing subsystems in partial designs. Alternative designs are the result of having multiple possible subsystem designs. NetSketch allows users to check whether any (or which) one of their alternative designs is safe (thus allowing the exploration of "what if" scenarios and tradeoffs), or whether every one of a set of possible deployments would be safe (thus establishing the safety of a system design subject to uncertainties regarding various settings in which the system may be deployed). Partial designs are the result of missing (*e.g.*, outsourced, or yet-to-be acquired) subsystems. These missing subsystems constitute "holes" in the system design. NetSketch enables users to infer the minimal requirements to be expected of (or imposed on) such holes. This enables the design of a system to proceed based only on promised functionality of missing parts thereof.

Formal analysis is at the heart of both of the above modeling and design activities. For example, in conjunction with a modeling activity in which the user identifies the boundaries of an interconnected set of modules that need to be encapsulated into a single subsystem, NetSketch must infer (through analysis) an appropriate set of constraints (*i.e.*, a typing) of that encapsulated subsystem. Similarly, in conjunction with a design activity in which the user specifies a subsystem as a set of alternative designs (or else as a hole), NetSketch must perform type checking (or type inference) to establish the safety of the design (or the minimal requirements expected of a subsystem that would fill the hole).

In a companion paper [7], we presented NetSketch from an operational perspective in support of modeling and design activities, by overviewing the processes it entails and illustrating its use in two applications: the management/shaping of traffic flows in a vehicular network (as a proxy for CPS applications) and in a streaming media network (as a proxy for Internet applications). In this paper, we focus on the more fundamental aspects of NetSketch – namely the formal system underlying its operation.

The NetSketch Formalism: Support for safety analysis in design and/or development tools such as NetSketch must be based on sound formalisms that are not specific to (and do not require expertise in) particular domains.[1] Not only should such formalisms be domain-agnostic, but also they must act as a unifying glue across multiple theories and calculi, allowing system integrators to combine (compose) exact results obtained through esoteric domain-specific techniques (*e.g.*, using network calculus to obtain worst-case delay envelopes, using scheduling theory to derive upper bounds on resource utilizations, or using control theory to infer convergence-preserving settings). This sort of approach lowers the bar for the expertise required to take full advantage of such domain-specific results at the small (sub-system) scale, while at the same time enabling scalability of safety analysis at the large (system) scale.

As we alluded before, NetSketch enables the composition of exact analyses of small subsystems by adopting a constrained-flow network formalism that exposes the tradeoffs between exactness of analysis and scalability of analysis. This is done using a strongly-typed Domain-Specific Language (DSL) for describing and reasoning about constrained-flow networks at various levels of "sketchiness" along with invariants that need to be enforced thereupon. In this paper, we formally define NetSketch's DSL and prove its soundness relative to appropriately-defined notions of validity.

A Motivating Example: Before delving into precise definitions and formal arguments, we outline the essential concepts that constitute our formalism for compositional analysis of problems involving constrainted-flow networks. We do so by considering (at a very high level) an example flow network systems problem in which compositional analysis of properties plays a role. Our goal is to identify essential aspects of these systems that we will later model precisely, and motivate their inclusion within the formalism. This example is considered in more precise detail in Section 7, and is also examined more extensively in a companion paper [7].

A software engineer in charge of developing a CPS vehicular traffic control application for a large metropolitan authority is faced with the following problem. Her city lies on a river bank across from the suburbs, and every morning hundreds of thousands of motorists drive across only a few bridges to get to work in the city center. Each bridge has a fixed number of lanes, but they are all reversible, enabling the application to determine how many lanes are available to inbound and outbound traffic during different times of the day. During morning rush hour, the goal of the system is to maximize the amount of traffic that can get into the city, subject to an overriding safety consideration – that no backups occur in the city center.

Modules and Networks: The city street grid is a network of a large number of only a few distinct kinds of traffic junctions (*e.g.*, forks, merges, and crossing junctions). Because the network is composed of many instances of a few modular components, if any analysis of the network is desired, it may be possible to take advantage of this modularity by analyzing the components individually in a more precise manner, and then composing the results to analyze the entire network. To this end, as detailed in Sections 2 and 3, our formalism provides means for defining *modules* (small network components) and assembling them into larger *networks* (graphs).

Constraints: Within our framework, analyses are represented using a language of *constraints*. If the engineer views

[1]While acceptable and perhaps expected for vertically-designed and smaller-scale (sub-)systems, deep domain expertise cannot be assumed for designers of horizontally-integrated, large-scale systems.

traffic as a flow across a network of modules, the relevant *parameters* describing this flow (*e.g.*, the number of open lanes, the density of traffic in the morning) can be mathematically constrained for each instance of a module. These constraints can model both the limitations of modules as well as the problem the engineer must solve. For example, a module corresponding to a merge junction may have two incoming lanes $1, 2$ and one outgoing lane 3, and the density of traffic travelling across the outgoing lane must be equal to the total traffic density travelling across the incoming lanes

$$d_1 + d_2 = d_3.$$

Likewise, constraints can model the problem to be solved. The engineer can find appropriate constraints for each of the three junction types that will ensure that no backups occur locally within that junction. For example, it may be the case for a junction that if the total density of entering traffic exceeds a "jam density" that makes the two entering traffics block each other, there will be backups. Thus, the engineer may choose to introduce a constraint such as

$$d_1 + d_2 \leqslant 10.$$

More complicated situations requiring the enforcement of additional desirable properties may introduce non-linear constraints. Once the local requirements are specified, a compositional analysis can answer interesting questions about the entire network, such as whether a configuration of lanes ensuring no backups is possible, or what the range of viable configurations may be.

Semantics and Soundness: So far, we have motivated the need for two intertwined languages: a language for describing networks composed of modules, and a language for describing constraints governing flows across the network components. But what precisely do the expressions in these languages mean, and how can we provide useful functionalities to the engineer, such as the ability to verify that constraints can be satisfied, to find solution ranges for these constraints, and to compose these analyses on modules to support analyses of entire networks? In order to ensure that our system works correctly "under the hood", it is necessary to define a precise *semantics* for these languages, along with a rigorous notion of what it means for an analysis of a network to be "correct". Only once these are defined is it possible to provide a guarantee that the system is indeed safe to use. To this end, we define a precise semantics for constraint sets and relationships between them, as well as network flows. In Section 8 we briefly sketch the proof of soundness for our formalism and refer readers to a complete proof of correctness in the full version of this paper [6].

2. MODULES: UNTYPED AND TYPED

We introduce several preliminary notions formally.

Definition 1. (*Syntax of Constraints*) We denote by $\mathbb{N}$ the set of natural numbers. The countably infinite set of *parameters* is $\mathcal{X} = \{x_0, x_1, x_2, \ldots\}$. The set of *constraints over* $\mathbb{N}$ *and* $\mathcal{X}$ can be defined in extended BNF style, where we use metavariables n and x to range over $\mathbb{N}$ and $\mathcal{X}$, respectively:

$$\begin{array}{lll} e \in \textsc{Exp} & ::= & n \mid x \mid e_1 * e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots \\ c \in \textsc{Const} & ::= & e_1 = e_2 \mid e_1 < e_2 \mid e_1 \leqslant e_2 \mid \ldots \end{array}$$

We include in Const at least equalities and orderings of expressions. Our examination can be extended to more general constraints, indicated by the ellipses "...", but the preceding give us enough to consider and to present our main ideas on compositional analysis. Possible extensions of Const include conditional constraints, negated constraints, time-dependent constraints, and others.

A special case of the constraints are the *linear constraints*, obtained by restricting the rule for Exp and Const:

$$\begin{array}{lll} e \in \textsc{LinExp} & ::= & n \mid x \mid n * x \mid e_1 + e_2 \\ c \in \textsc{LinConst} & ::= & e_1 = e_2 \mid e_1 < e_2 \mid e_1 \leqslant e_2 \end{array}$$

In what follows, constraints in Const are part of a *given* flow network abstraction and may be arbitrarily complex; constraints in LinConst are to be *inferred* and/or *checked* against the given constraints. Constraints in LinConst are hopefully simple enough so that their manipulation does not incur a prohibitive cost, but expressive enough so that their satisfaction guarantee desirable properties of the flow network under exmination.

Depending on the application, the set $\mathcal{X}$ of parameters may be n-sorted for some finite $n \geqslant 1$. For example, in relation to vehicular traffic networks, we may choose $\mathcal{X}$ to be 2-sorted, one sort for *velocity parameters* and one sort for *density parameters.*

When there are several sorts, dimensionality restrictions must be heeded. For traffic networks with two sorts, the velocity dimension is *unit distance/unit time*, *e.g.*, *kilometer/hour*, and the density dimension is *unit mass/unit distance*, *e.g.*, *ton/kilometer.* Thus, multiplying a velocity v by a density d produces a quantity $v * d$, namely a *flow*, which is measured in *unit mass/unit time*, *e.g.*, *ton/hour.* If we add two expressions e_1 and e_2, or subtract them, or compare them, then e_1 and e_2 must have the same dimension, otherwise the resulting expression is meaningless.

In the abstract setting of our examination below we do not need to worry about such restrictions on expressions: they will be implicitly satisfied by our constraints if they correctly model the behavior of whatever networks are under consideration.

Definition 2. (*Untyped Modules*) We specify an untyped module $\mathcal{A}$ by a four-tuple: $(\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})$ where:

$$\begin{array}{lll} \mathcal{A} & = & \text{name of the module} \\ \mathsf{In} & = & \text{finite set of input parameters} \\ \mathsf{Out} & = & \text{finite set of output parameters} \\ \mathsf{Con} & = & \text{finite set of constraints over } \mathbb{N} \text{ and } \mathcal{X} \end{array}$$

where $\mathsf{In} \cap \mathsf{Out} = \varnothing$ and $\mathsf{In} \cup \mathsf{Out} \subseteq \mathsf{parameters}(\mathsf{Con})$, where $\mathsf{parameters}(\mathsf{Con})$ is the set of parameters occurring in Con.

We are careful in adding the name of the module, $\mathcal{A}$, to its specification; in the formal setup of Section 3, we want to be able to refer to the module by its name without the overhead of the rest of its specification. By a slight abuse of notation, we may write informally: $\mathcal{A} = (\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})$. Thus, "$\mathcal{A}$" may refer to the full specification of the module or may be just its name.

We use upper-case calligraphic letters to refer to modules and networks – from the early alphabet ($\mathcal{A}$ and $\mathcal{B}$) for modules and from the middle alphabet ($\mathcal{M}$,$\mathcal{N}$ and $\mathcal{P}$) for networks.

Definition 3. (*Typed Modules*) Consider a module $\mathcal{A}$ as specified in Definition 2. A *typing judgment*, or a *typed specification*, or just a *typing*, for $\mathcal{A}$ is an expression of the form

($\mathcal{A}$: Con^*), where Con^* is a finite set of *linear* constraints over $\mathsf{In} \cup \mathsf{Out}$. As it stands, a typing judgment ($\mathcal{A}$: Con^*) may or may not be valid. The validity of judgments presumes a formal definition of the semantics of modules, which we introduce in Section 4.

To distinguish between a constraint in Con, which is arbitrarily complex, and a constraint in Con^*, which is always linear, we refer to the former as "given" or "internal" and to the latter as a "type".

3. NETWORK SKETCHES: UNTYPED

We define a specification language to assemble modules together, also allowing for the presence of network holes. This is a strongly-typed *d*omain-*s*pecific *l*anguage (DSL), which can be used in two modes, with and without the types inserted. Our presentation is in two parts, the first without types and the second with types. In this section, we present the first part, when our DSL is used to construct networks without types inserted. In Section 6, we re-define our DSL with types inserted. This two-part presentation allows us to precisely define the difference between "untyped specification" and "typed specification" of a flow network.

"Network holes" are place-holders. We later attach some attributes to network holes (they are not totally unspecified), in Definitions 6 and 13. We use X, Y, and Z, possibly decorated, to denote network holes.

An untyped network sketch is written as $(\mathcal{M}, I, O, \mathcal{C})$, where I and O are the sets of input and output parameters, and $\mathcal{C}$ is a finite set of finite constraint sets.[2] $\mathcal{M}$ is *not* a name but an expression built up from: (1) module names, (2) hole names, and (3) the constructors **conn**, **loop** and **let-in**.[3] Nevertheless, we may refer to such a sketch by just writing the expression $\mathcal{M}$, and by a slight abuse of notation we may also write $\mathcal{M} = (\mathcal{M}, I, O, \mathcal{C})$. For such an untyped network $\mathcal{M}$, we define $\mathsf{In}(\mathcal{M})$ as I (the set of input parameters) and $\mathsf{Out}(\mathcal{M})$ as O (the set of output parameters).

Definition 4. (*Syntax of Untyped Network Sketches*) In extended BNF style:

$$
\begin{array}{lll}
\mathcal{A},\mathcal{B},\mathcal{C} & \in \textsc{ModuleNames} & \\
X,Y,Z & \in \textsc{HoleNames} & \\
\mathcal{M},\mathcal{N},\mathcal{P} & \in \textsc{RawSketches} \ ::= & \\
& \quad \mathcal{A} & \\
& \mid X & \\
& \mid \mathbf{conn}(\theta,\mathcal{M},\mathcal{N}) & \theta \subseteq_{1\text{-}1} \mathsf{Out}(\mathcal{M}) \times \mathsf{In}(\mathcal{N}) \\
& \mid \mathbf{loop}(\theta,\mathcal{M}) & \theta \subseteq_{1\text{-}1} \mathsf{Out}(\mathcal{M}) \times \mathsf{In}(\mathcal{M}) \\
& \mid \mathbf{let}\ X \in \{\mathcal{M}_1,\ldots,\mathcal{M}_n\}\ \mathbf{in}\ \mathcal{N} & X \text{ occurs once in } \mathcal{N}
\end{array}
$$

We write $\theta \subseteq_{1\text{-}1} \mathsf{Out}(\mathcal{M}) \times \mathsf{In}(\mathcal{N})$ to denote a partial one-one map from $\mathsf{Out}(\mathcal{M})$ to $\mathsf{In}(\mathcal{N})$. (If the set of parameters is sorted with more than one sort – for example, *velocity* and *density* – then θ must respect sorts, *i.e.*, if $(x, y) \in \theta$ then x and y are either both velocity parameters or both density parameters.)

The formal expressions written according to the preceding BNF are said to be "raw" because they do not specify how the internal constraints of a network sketch are assembled together from those of its subcomponents. This is what the rules in Figure 1 do precisely.

In an expression "**let** $X \in \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ **in** $\mathcal{N}$ ", we call "$X \in \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$" a *binding* for the hole X and "$\mathcal{N}$" the *scope* of this binding. Informally, the idea is that all of the network sketches in $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ can be interchangeably placed in the hole X, depending on changing conditions of operation in the network as a whole. If a hole X occurs in a network sketch $\mathcal{M}$ outside the scope of any **let**-binding, we say X is *free* in $\mathcal{M}$. If there are no free occurrences of holes in $\mathcal{M}$, we say that $\mathcal{M}$ is *closed*.

Note carefully that $\mathcal{M}, \mathcal{N}$ and $\mathcal{P}$ are *metavariables*, ranging over expressions in $\textsc{RawSketches}$; they do not appear as formal symbols in such expressions written in full. By contrast, $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ are *names* of modules and can occur as formal symbols in expressions of $\textsc{RawSketches}$. $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ are like names of "prim ops" in well-formed phrases of a programming language.

In the examination to follow, we want each occurrence of the same module or the same hole in a specification to have its own private set of names, which we achieve using isomorphic renaming.

Definition 5. (*Fresh Isomorphic Renaming*) Let A be an object defined over parameters. Typically, A is a module or a network sketch. Suppose the parameters in A are called $\{x_1, x_2, \ldots\}$. We write $'A$ to denote the same object A, whose name is also $'A$ and with all parameter names freshly renamed to $\{'x_1, 'x_2, \ldots\}$. We want these new names to be fresh, *i.e.*, nowhere else used and private to $'A$. Thus, A and $'A$ are isomorphic but distinct objects.

Sometimes we need two or more isomorphic copies of A in the same context. We may therefore consider $'A$ and $''A$. If there are more than two copies, it is more convenient to write ${}^1A$, ${}^2A$, ${}^3A$, etc.

We also need to stipulate that, given any of the isomorphic copies of object A, say ${}^nA$, we can retrieve the original A, along with all of its original names, from ${}^nA$.

There are other useful constructs in the DSL of Definition 4. But these will be either special cases of the basic three constructs – **conn**, **loop**, and **let-in** – or macros which can be "de-sugared" into expressions only involving the basic three. One important macro is the **let-in** construct where the hole occurs several times in the scope, instead of just once:

$$\mathbf{let}^*\ X \in \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}\ \mathbf{in}\ \mathcal{N} \qquad X \text{ occurs } q \geqslant 1 \text{ times in } \mathcal{N}$$

To analyze the preceding expression, using the typing rules in this section, we de-sugar in a particular way:

$$
\begin{array}{l}
\mathbf{let}\ \ {}^1X \in \{{}^1\mathcal{M}_1, \ldots, {}^1\mathcal{M}_n\}\ \ \mathbf{in} \\
\mathbf{let}\ \ {}^2X \in \{{}^2\mathcal{M}_1, \ldots, {}^2\mathcal{M}_n\}\ \ \mathbf{in} \\
\ \ldots \\
\mathbf{let}\ \ {}^qX \in \{{}^q\mathcal{M}_1, \ldots, {}^q\mathcal{M}_n\}\ \ \mathbf{in}\ \ \mathcal{N}[X^{(1)} := {}^1X, \ldots, X^{(q)} := {}^qX]
\end{array}
$$

where $X^{(1)}, \ldots, X^{(q)}$ denote the q occurrences of X in $\mathcal{N}$ (the superscripts are not part of the syntax, just bookkeeping notation for this explanation), ${}^1X, \ldots, {}^qX$ are fresh distinct hole names, and $\{{}^p\mathcal{M}_1, \ldots, {}^p\mathcal{M}_n\}$ is a fresh isomorphic copy of $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ for every $1 \leqslant p \leqslant q$.

[2]Note $\mathcal{C}$ is a set of constraint sets, not a single constraint set. This allows for placing different network sketches into the same hole. Each constraint set in $\mathcal{C}$ corresponds to one way of filling all the holes in $\mathcal{M}$.

[3]We may customize or add other constructors according to need. There are also alternative constructors. For example, instead of **conn**, we may introduce **par** (for *parallel* composition of two network sketches), and then "de-sugar" **conn** as a combination of a single **par** followed by a single **loop**. Conversely, **par** can be expressed using **conn**, the former is a special case of the latter when $\theta = \varnothing$. Our choice of constructors here is for didactic and expository reasons.

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

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

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

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

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

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

(where φ and ψ are isomorphisms, different for different values of k)

Figure 1: Rules for Untyped Network Sketches.

Definition 6. (*Untyped Network Holes*) An *untyped network hole* is a triple: $(X,\mathsf{In},\mathsf{Out})$ where X is the name of the hole, In is a finite set of input parameters, and Out is a finite set of output parameters. As usual, for the sake of brevity we sometimes write: $X\ =\ (X,\mathsf{In},\mathsf{Out})$

There are 5 inference rules: Module, Hole, Connect, Loop, and Let, one for each of the 5 cases in the BNF in Definition 4. These are shown in Figure 1.

The renaming in rule Module is to insure that each occurrence of the same module has its own private names of paramaters. In rule Hole we do not need to rename, because there will be exactly one occurrence of each hole, whether bound or free, each with its own private set of names.

Rule Connect takes two network sketches, $\mathcal{M}$ and $\mathcal{N}$, and returns a network sketch $\mathbf{conn}(\theta,\mathcal{M},\mathcal{N})$ where some of the output parameters in $\mathcal{M}$ are unified with some of the input parameters in $\mathcal{N}$, according to what θ prescribes.

Rule Loop takes one network sketch, $\mathcal{M}$, and returns a new network sketch $\mathbf{loop}(\theta,\mathcal{M})$ where some of the output parameters in $\mathcal{M}$ are identified with some of the input parameters in $\mathcal{M}$ according to θ.

Rule Let is a little more involved than the preceding rules. The complication is in the way we define the collection $\mathcal{C}'$ of constraint sets in the conclusion of the rule. Suppose $\mathcal{C}_k=\{C_{k,1},C_{k,2},\ldots,C_{k,s(k)}\}$, *i.e.*, the flow through $\mathcal{M}_k$ can be regulated according to $s(k)$ different constraint sets, for every $1\leqslant k\leqslant n$. The definition of the new collection $\mathcal{C}'$ of constraint sets should be read as follows: For every $\mathcal{M}_k$, for every possible way to regulate the flow through $\mathcal{M}_k$ (*i.e.*, for every possible $r\in\{1,\ldots,s(k)\}$), for every way of placing network $\mathcal{M}_k$ in hole X (*i.e.*, every isomorphism (φ,ψ) from (I_k,O_k) to $(\mathsf{In},\mathsf{Out})$), add the corresponding constraint set to the collection $\mathcal{C}'$.

In the side-condition of rule Let, the maps φ and ψ are isomorphisms. If parameters are multi-sorted, then φ and ψ must respect sorts, *i.e.*, if $\varphi(x)=y$ then both x and y must be of the same sort, *e.g.*, both velocity parameters, or both density parameters, etc., and similarly for ψ.

In particular applications, we may want the placing of $\mathcal{M}_k$ in hole X to be uniquely defined for every $1\leqslant k\leqslant n$, rather than multiply-defined in as many ways as there are isomorphism pairs from (I_k,O_k) to $(\mathsf{In},\mathsf{Out})$. For this, we may introduce structured parameters, *i.e.*, finite sequences of parameters, and also restrict the network hole X to have one (structured) input parameter and one (structured) output parameter. This requires the introduction of selectors, which allow the retrieval of individual parameters from a sequence of parameters.[4]

4. SEMANTICS OF NETWORK TYPINGS

A network typing, as later defined in Section 6, is specified by an expression of the form $(\mathcal{M},I,O,\mathcal{C})\,:\,C^*$ where $(\mathcal{M},I,O,\mathcal{C})$ is an untyped network and C^* is a finite set of linear constraints such that $\mathsf{parameters}(C^*)\subseteq I\cup O$.[5]

Definition 7. (*Satisfaction of Constraints*) Let $\mathcal{Y}\subseteq\mathcal{X}$, a subset of parameters. Let $\textsc{val}$ be a *valuation for* $\mathcal{Y}$, *i.e.*, $\textsc{val}$ is a map from $\mathcal{Y}$ to $\mathbb{N}$. We use "$\models$" to denote the satisfaction relation. Let C be a finite set of constraints such that $\mathsf{parameters}(C)\subseteq\mathcal{Y}$. *Satisfaction* of C by $\textsc{val}$ is defined in the usual way and written $\textsc{val}\models C$.

Definition 8. (*Closure of Constraint Sets*) Let $\mathcal{Y}\subseteq\mathcal{X}$. Let C and C' be constraint sets over $\mathbb{N}$ and $\mathcal{Y}$. We say that C *implies* C' iff, for every valuation $\textsc{val}:\mathcal{Y}\to\mathbb{N}$,

$$\textsc{val}\models C\quad\text{implies}\quad\textsc{val}\models C'.$$

If C implies C', we write $C\ \Rightarrow\ C'$. For a finite constraint set C, its *closure* is the set of all constraints implied by C, namely, $\mathsf{closure}(C)=\{c\in\textsc{Const}\,|\,C\ \Rightarrow\ \{c\}\}$.

In general, $\mathsf{closure}(C)$ is an infinite set. We only consider infinite constraint sets that are the closures of finite sets of linear constraints. Following standard terminology,

[4]This variation of rule Let, where there is a unique way of inserting every $\mathcal{M}_k$ with $1\leqslant k\leqslant n$ in the hole X, corresponds also to the situation when the system designer can control the "wiring" of every $\mathcal{M}_k$ in X and wants it to be uniquely determined.

[5]Note that $\mathsf{parameters}(C^*)\subseteq I\cup O\subseteq\mathsf{parameters}(\mathcal{C})$, *i.e.*, all the parameters in the boundary constraints C^* (aka types) are in $I\cup O$, and $I\cup O$ is a subset of all the parameters in the internal constraints $\mathcal{C}$.

such an infinite constraint set is said to have a *finite basis*.[6] In actual applications, we are interested in "minimal" finite bases that do not contain "redundant" constraints. It is reasonable to define a "minimal finite basis" for a constraint set if it is smallest in size. The problem is that minimal bases in this sense are not uniquely defined. How to compute minimal finite bases, and how to uniquely select a canonical one among them, are issues addressed by an implementation.

Let C be a constraint set and A a set of parameters. We define two restrictions of C relative to A:

$$\begin{aligned} C \upharpoonright A &= \{\, c \in C \mid \mathsf{parameters}(c) \subseteq A \,\}, \\ C \downharpoonright A &= \{\, c \in C \mid \mathsf{parameters}(c) \cap A \neq \varnothing \,\}. \end{aligned}$$

That is, $(C \upharpoonright A)$ is the set of constraints in C where only parameters from A occur, and $(C \downharpoonright A)$ is the set of constraints in C with at least one occurrence of a parameter from A.

We introduce two different semantics, corresponding to what we call "weak satisfaction" and "strong satisfaction" of typing judgements. Both semantics are meaningful, corresponding to whether or not network nodes act as "autonomous systems", *i.e.*, whether or not each node coordinates its action with its neighbors or according to instructions from a network administrator.

Definition 9. (*Weak and Strong Satisfaction*) Let $\mathcal{M} = (\mathcal{M}, I, O, \mathcal{C})$ be an untyped network sketch and $(\mathcal{M} : C^*)$ a typing for $\mathcal{M}$. Recall that $\mathsf{parameters}(C^*) \subseteq I \cup O$. We partition $\mathsf{closure}(C^*)$ into two subsets as follows:

$$\begin{aligned} \mathsf{pre}(C^*) &= \mathsf{closure}(C^*) \upharpoonright I \\ \mathsf{post}(C^*) &= \mathsf{closure}(C^*) - \mathsf{pre}(C^*) = \mathsf{closure}(C^*) \downharpoonright O \end{aligned}$$

The "pre( )" is for "pre-conditions" and the "post( )" is for "post-conditions". While the parameters of $\mathsf{pre}(C^*)$ are all in I, the parameters of $\mathsf{post}(C^*)$ are not necessarily all in O, because some constraints in C^* may contain both input and output parameters.[7]

The definitions of "weak satisfaction" and "strong satisfaction" below are very similar except that the first involves an *existential quantification* and the second a *universal quantification*. We use "$\models_{\mathrm{w}}$" and "$\models_{\mathrm{s}}$" to denote weak and strong satisfaction. For the rest of this definition, let $\textsc{val}$ be a fixed valuation of the input parameters of $\mathcal{M}$, $\textsc{val} : I \to \mathbb{N}$.

We say that $\textsc{val}$ ***weakly satisfies*** $(\mathcal{M} : C^*)$ and write $\textsc{val} \models_{\mathrm{w}} (\mathcal{M} : C^*)$ to mean that if

- $\textsc{val} \models \mathsf{pre}(C^*)$

then for every $C \in \mathcal{C}$ ***there is a valuation*** $\textsc{val}' \supseteq \textsc{val}$ such that both of the following conditions are true:

- $\textsc{val}' \models C$
- $\textsc{val}' \models \mathsf{post}(C^*)$

[6]If we set up a logical system of inference for our linear constraints, using some kind of equational reasoning, then an infinite constraint set has a "finite basis" iff it is "finitely axiomatizable".

[7]Both $\mathsf{pre}(C^*)$ and $\mathsf{post}(C^*)$ are infinite sets. In the abstract setting of this report, this is not a problem. In an actual implementation, we need an efficient method for computing "minimal finite bases" for $\mathsf{pre}(C^*)$ and $\mathsf{post}(C^*)$, or devise an efficient algorithm to decide whether a constraint is in one of these sets.

Informally, $\textsc{val}$ weakly satisfies $(\mathcal{M} : C^*)$ when: **if** $\textsc{val}$ satisfies $\mathsf{pre}(C^*)$, **then** there is an extension $\textsc{val}'$ of $\textsc{val}$ satisfying the internal constraints of $\mathcal{M}$ **and** $\mathsf{post}(C^*)$.

We say that $\textsc{val}$ ***strongly satisfies*** $(\mathcal{M} : C^*)$ and write $\textsc{val} \models_{\mathrm{s}} (\mathcal{M} : C^*)$ to mean that if

- $\textsc{val} \models \mathsf{pre}(C^*)$

then for every $C \in \mathcal{C}$ and ***every valuation*** $\textsc{val}' \supseteq \textsc{val}$, if

- $\textsc{val}' \models C$

then the following condition is true:

- $\textsc{val}' \models \mathsf{post}(C^*)$

Informally, $\textsc{val}$ strongly satisfies $(\mathcal{M} : C^*)$ when: **if** $\textsc{val}$ satisfies $\mathsf{pre}(C^*)$ **and** $\textsc{val}'$ is an extension of $\textsc{val}$ satisfying the internal constraints of $\mathcal{M}$, **then** $\textsc{val}'$ satisfies $\mathsf{post}(C^*)$.

Definition 10. (*Weak and Strong Validity of Typings*) Let $(\mathcal{M} : C^*)$ be a typing for network $\mathcal{M} = (\mathcal{M}, I, O, \mathcal{C})$. We say that $(\mathcal{M} : C^*)$ is *weakly valid* – resp. *strongly valid* – iff for every valuation $\textsc{val} : \mathsf{parameters}(\mathsf{pre}(C^*)) \to \mathbb{N}$, it holds that $\textsc{val} \models_{\mathrm{w}} (\mathcal{M} : C^*)$ – resp. $\textsc{val} \models_{\mathrm{s}} (\mathcal{M} : C^*)$. If $(\mathcal{M} : C^*)$ is weakly valid, we write $\textsc{val} \models_{\mathrm{w}} (\mathcal{M} : C^*)$, and if strongly valid, we write $\textsc{val} \models_{\mathrm{s}} (\mathcal{M} : C^*)$.

Informally, $(\mathcal{M} : C^*)$ is *weakly valid* under the condition that, for every network flow satisfying $\mathsf{pre}(C^*)$, ***there is a way*** of channelling the flow through $\mathcal{M}$, consistent with its internal constraints, so that $\mathsf{post}(C^*)$ is satisfied. And $(\mathcal{M} : C^*)$ is *strongly valid* under the condition that, for every network flow satisfying $\mathsf{pre}(C^*)$ and ***for every way*** of channelling the flow through $\mathcal{M}$, consistent with its internal constraints, $\mathsf{post}(C^*)$ is satisfied.

5. ORDERING OF NETWORK TYPINGS

We define a precise way of deciding that a typing is "stronger" (or "more informative") or "weaker" (or "less informative") than another typing.

Definition 11. (*Comparing Typings*) Let $\mathcal{M} = (\mathcal{M}, I, O, \mathcal{C})$ be a untyped network sketch and let $(\mathcal{M} : C^*)$ a typing for $\mathcal{M}$. We use again the notions of "preconditions" and "postconditions" from Definition 9, but to make explicit that these relate to $\mathcal{M}$, we write $\mathsf{pre}(\mathcal{M} : C^*)$ instead of $\mathsf{pre}(C^*)$ and $\mathsf{post}(\mathcal{M} : C^*)$ instead of $\mathsf{post}(C^*)$, resp.

Let $(\mathcal{M} : C_1^*)$ and $(\mathcal{M} : C_2^*)$ be two typings for the same network sketch $\mathcal{M}$. We say $(\mathcal{M} : C_1^*)$ *implies* – or *is more precise than* – $(\mathcal{M} : C_2^*)$ and we write: $(\mathcal{M} : C_1^*) \Rightarrow (\mathcal{M} : C_2^*)$ just in case the two following conditions hold:

1. $\mathsf{pre}(\mathcal{M} : C_1^*) \Leftarrow \mathsf{pre}(\mathcal{M} : C_2^*)$, *i.e.*, the precondition of $(\mathcal{M} : C_1^*)$ is weaker than that of $(\mathcal{M} : C_2^*)$.
2. $\mathsf{post}(\mathcal{M} : C_1^*) \Rightarrow \mathsf{post}(\mathcal{M} : C_2^*)$, *i.e.*, the postcondition of $(\mathcal{M} : C_1^*)$ is stronger than that of $(\mathcal{M} : C_2^*)$.

We say $(\mathcal{M} : C_1^*)$ and $(\mathcal{M} : C_2^*)$ are *equivalent*, and write: $(\mathcal{M} : C_1^*) \Leftrightarrow (\mathcal{M} : C_2^*)$ in case $(\mathcal{M} : C_1^*) \Rightarrow (\mathcal{M} : C_2^*)$ *and* $(\mathcal{M} : C_1^*) \Leftarrow (\mathcal{M} : C_2^*)$. If $(\mathcal{M} : C_1^*) \Leftrightarrow (\mathcal{M} : C_2^*)$, it does not necessarily follow that $C_1^* = C_2^*$, because constraints implying each other are not necessarily identical.

Normally we are interested in deriving "optimal" network typings, which are the most informative about the flows that the network can safely handle. We can also call them "minimal" rather than "optimal" because we think of

$$\textsc{Hole}\qquad \frac{(X,\mathsf{In},\mathsf{Out}) : \mathsf{Con}^* \ \in\ \Gamma}{\Gamma \vdash (X,\mathsf{In},\mathsf{Out},\{\ \}) : \mathsf{Con}^*}$$

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

$$\textsc{Connect}\qquad \frac{\Gamma \vdash (\mathcal{M},I_1,O_1,\mathcal{C}_1) : C_1^* \qquad \Gamma \vdash (\mathcal{N},I_2,O_2,\mathcal{C}_2) : C_2^*}{\Gamma \vdash (\mathbf{conn}(\theta,\mathcal{M},\mathcal{N}),I,O,\mathcal{C}) : C^*}$$

$$\theta \subseteq_{\text{1-1}} O_1 \times I_2,\ I = I_1 \cup (I_2 - \mathsf{ran}(\theta)),\ O = (O_1 - \mathsf{dom}(\theta)) \cup O_2,\ C^* = (C_1^* \cup C_2^*) \upharpoonright (I \cup O),$$

$$\text{(Ct)}\quad \boxed{\mathsf{post}(\mathcal{M} : C_1^*) \Rightarrow \{x = y \mid (x,y) \in \theta\} \cup (\mathsf{pre}(\mathcal{N} : C_2^*) \downharpoonleft \mathsf{ran}(\theta))}$$

$$\textsc{Loop}\qquad \frac{\Gamma \vdash (\mathcal{M},I_1,O_1,\mathcal{C}_1) : C_1^*}{\Gamma \vdash (\mathbf{loop}(\theta,\mathcal{M}),I,O,\mathcal{C}) : C^*}$$

$$\theta\ \subseteq_{\text{1-1}} O_1 \times I_1,\ \ I = I_1 - \mathsf{ran}(\theta),\ \ O = O_1 - \mathsf{dom}(\theta),\ \ C^* = C_1^* \upharpoonright (I \cup O),$$

$$\text{(Lp)}\quad \boxed{\mathsf{post}(\mathcal{M} : C_1^*) \Rightarrow \{x = y \mid (x,y) \in \theta\} \cup (\mathsf{pre}(\mathcal{M} : C_1^*) \downharpoonleft \mathsf{ran}(\theta))}$$

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

for all $1 \leqslant k \leqslant n$ and pairs of bijections $(\varphi,\psi) : (I_k,O_k) \to (\mathsf{In},\mathsf{Out})$:

$$\text{(Lt)}\quad \boxed{C_k^* \ \Leftrightarrow\ \big(\mathsf{Con}^* \cup \{\, x = \varphi(x) \mid x \in I_k\,\} \cup \{\, x = \psi(x) \mid x \in O_k\,\}\big)}$$

$$\textsc{Weaken}\qquad \frac{\Gamma \vdash (\mathcal{M},I,O,\mathcal{C}) : C_1^*}{\Gamma \vdash (\mathcal{M},I,O,\mathcal{C}) : C^*} \quad \text{(Wn)}\quad \boxed{\mathsf{pre}(\mathcal{M} : C_1^*) \Leftarrow \mathsf{pre}(\mathcal{M} : C^*) \ \text{ and }\ \mathsf{post}(\mathcal{M} : C_1^*) \Rightarrow \mathsf{post}(\mathcal{M} : C^*)}$$

Figure 2: Rules for Typed Network Sketches.

them as being "at the bottom" of a partial ordering on typings. This is analogous to the *principal* (or *most general*) type of a function in a strongly-typed functional programming language; the principal type is the bottom element in the lattice of valid types for the function. This analogy shouldn't be pushed too far, however; a principal type is usually unique, whereas optimal typings are usually multiple.

Definition 12. (*Optimal Typings*) Let $(\mathcal{M} : C_1^*)$ be a typing for a network sketch $\mathcal{M}$. We say $(\mathcal{M} : C_1^*)$ is an *optimal weakly-valid typing* just in case:

- $(\mathcal{M} : C_1^*)$ is a weakly-valid typing.
- For every weakly-valid typing $(\mathcal{M} : C_2^*)$, if $(\mathcal{M} : C_2^*) \Rightarrow (\mathcal{M} : C_1^*)$ then $(\mathcal{M} : C_2^*) \Leftrightarrow (\mathcal{M} : C_1^*)$.

Define *optimal strongly-valid typing* similarly, with "strongly" substituted for "weakly" in the two preceding bullet points.

6. NETWORK SKETCHES: TYPED

We define typed specifications by the same inference rules we already used to derive untyped specifications in Section 3, but now augmented with type information.

Definition 13. (*Typed Network Holes*) This continues Definition 6. The network hole $(X,\mathsf{In},\mathsf{Out})$ is *typed* if it is supplied with a finite set of *linear* constraints Con^* – *i.e.*, a type – written over $\mathsf{In} \cup \mathsf{Out}$. A fully specified *typed network hole* is written as "$(X,\mathsf{In},\mathsf{Out}) : \mathsf{Con}^*$".

For simplicity, we may refer to $(X,\mathsf{In},\mathsf{Out})$ by its name X and write $(X : \mathsf{Con}^*)$ instead of $(X,\mathsf{In},\mathsf{Out}) : \mathsf{Con}^*$ with the understanding that the omitted attributes can be uniquely retrieved by reference to the name X of the hole.

We repeat the rules Module, Hole, Connect, Loop, and Let, with the type information inserted. As they elaborate the previous rules, we omit some of the side conditions; we mention only the parts that are necessary for inserting the typing. The rules are shown in Figure 2.

In each of the rules, we highlight the crucial side-condition by placing it in a framed box; this condition expresses a relationship that must be satisfied by the "derived types" (linear constraints) in the premises of the rule. For later reference, we call this side-condition (Ct) in Connect, (Lp) in Loop, and (Lt) in Let.

There are different versions of rule Let depending on the side condition (Lt) – the weaker the side condition, the more powerful the rule, *i.e.*, the more network sketches for which it can derive a typing. The simplest way of formulating Let, as shown in Figure 2, makes the side condition most restrictive.

However, if we introduce the rule Weaken, the last shown in Figure 2, the side condition (Lt) is far less restrictive than it appears; it allows to adjust the derived types and constraints of the networks in $\{\mathcal{M}_1,\ldots,\mathcal{M}_n\}$ in order to satisfy (Lt), if possible by weakening them. (The rule Weaken plays the same role as a *subtyping* rule in the type system of an object-oriented programming language.)

7. EXAMPLE

This example is a follow-up to one of the use cases (vehicular traffic) in our companion report [7], and falls within the context of the vehicular traffic problem we discussed at a high level in the introduction. We consider a module $\mathcal{A}$ whose untyped specification is defined by a set Con of internal constraints over input parameters $\{d_1,d_2,d_3\}$ and output parameters $\{d_4,d_5,d_6\}$. In this particular module $\mathcal{A}$, there are no purely internal parameters, *i.e.*, all are either input or output parameters. Con consists of:

(a)	$2 \leqslant d_1, d_4, d_5, d_6 \leqslant 8$	bounds on d_1, d_4, d_5 and d_6
(b)	$0 \leqslant d_2, d_3 \leqslant 6$	bounds on d_2 and d_3
(c)	$d_1 = d_4 + d_5$	constraint at node A
(d)	$d_2 + d_3 + d_4 \leqslant 10$	constraint at node B
(e)	$d_2 + d_3 + d_5 \leqslant 10$	constraint at node C
(f)	$d_2 + d_3 = d_6$	constraint at node C

In this simple example, all the constraints in Con are linear. Nevertheless, many of the issues and complications we need to handle with non-linear internal constraints already arise here. In the complete version of this report [6], we discuss the following issues for this particular example, illustrating what an implementation has to deal in full generality:

- Alternative methods of inferring *weak* and *strong* typings for $\mathcal{A}$, and how to make these methods more efficient computationally.
- Efficiently deciding whether a typing for $\mathcal{A}$, weak or strong, is *optimal*.
- Inferring typings for $\mathcal{A}$ relative to *objective functions* such as: maximizing the sum of the three input flows (at parameters d_1, d_2, d_3), maximizing and equalizing all three input flows, equalizing the flow at a particular input (*e.g.*, at d_1) with that at a particular output (*e.g.*, at d_6), etc.
- How to handle and choose between several *weakest pre-conditions* for the same post-condition for $\mathcal{A}$, or between several *strongest post-conditions* for the same pre-condition for $\mathcal{A}$.[8]
- Some of the limitations when we switch from *base mode* (or whole-system analysis of modules) to *sketch mode* (or compositional analysis of network sketches, using the rules of Sections 3 and 6).

Briefly here, we consider a simple network sketch $\mathcal{N}$ assembled from module $\mathcal{A}$ and network hole X. Suppose X is assigned two input $\{i_1, i_2\}$ and two output parameters $\{o_1, o_2\}$. The untyped version of $\mathcal{N}$ is:

$$\textsf{let } X \in \Big\{\textbf{loop}(\{(d_6, d_1)\}, \mathcal{A})\Big\} \textsf{ in } \textbf{conn}\Big(\{(o_1, {'d_3}), (o_2, {'d_2})\}, X, {'\mathcal{A}}\Big)$$

Graphic representations of $\mathcal{A}$, (from [7]), "$\textbf{loop}(\{(d_6, d_1)\}, \mathcal{A})$" and "$\textbf{conn}(\{(o_1, {'d_3}), (o_2, {'d_2})\}, X, {'\mathcal{A}})$" – but not $\mathcal{N}$ – are shown in Figure 3. ${'\mathcal{A}}$ is an isomorphic copy of $\mathcal{A}$ with its own fresh set of parameters $\{{'d_1}, {'d_2}, {'d_3}, {'d_4}, {'d_5}, {'d_6}\}$.

Note that for $\mathcal{N}$ there are four possible ways of placing "$\textbf{loop}(\{(d_6, d_1)\}, \mathcal{A})$" in the hole X, because there are two possible isomorphisms between the input parameters and two possible isomorphisms between the output parameters, for a total of 4 possible isomorphism pairs

$$(\varphi, \psi) : (\{d_2, d_3\}, \{d_4, d_5\}) \to (\{i_1, i_2\}, \{o_1, o_2\})$$

See the side condition of rule Let in Figures 1 and 2.

[8]Contrary to the situation with, say, Hoare logic for imperative programs, we can have several mutually incomparable weakest pre-conditions for the same post-condition, as well as several mutually incomparable strongest post-conditions for the same pre-condition, for a given module or network sketch.

In the full version of this report [6], relative to the objective function $d_1 = d_6$, we infer the following strong typing/linear constraints Con^* for $\mathcal{A}$, where we write "$x : [4, 6]$" instead of "$4 \leqslant x \leqslant 6$" to save space:

$$\begin{aligned}\mathsf{Con}^* \;=\; \{\; & d_1 : [4,6],\; d_2 : [2,3],\; d_3 : [2,3],\\ & d_4 : [2,3],\; d_5 : [2,3],\; d_6 : [4,6],\\ & d_1 = d_2 + d_3,\; d_1 = d_6,\; d_2 + d_3 = d_4 + d_5 \;\}\end{aligned}$$

For this typing $(\mathcal{A} : \mathsf{Con}^*)$, we have:

$$\begin{aligned}&\mathsf{pre}(\mathcal{A} : \mathsf{Con}^*) \;\supseteq \{d_1{:}[4,6],\; d_2{:}[2,3],\; d_3{:}[2,3],\; d_1 = d_2 + d_3\}\\ &\mathsf{post}(\mathcal{A} : \mathsf{Con}^*) = \mathsf{closure}(\mathsf{Con}^*) - \mathsf{pre}(\mathcal{A} : \mathsf{Con}^*)\end{aligned}$$

$\mathsf{pre}(\mathcal{A} : \mathsf{Con}^*)$ includes other constraints besides those listed, obtained by taking their closure.

Switching to *sketch mode*, starting from the preceding typing $(\mathcal{A} : \mathsf{Con}^*)$, we can check that side condition (Lp) of rule Loop in Figure 2 is satisfied (verification omitted here). This allows to derive the strong typing:

$$\mathsf{Con}_1^* =$$
$$\{d_2 : [2,3],\; d_3 : [2,3], d_4 : [2,3],\; d_5 : [2,3],\; d_2 + d_3 = d_4 + d_5\}$$

for the network sketch "$\textbf{loop}(\{(d_6, d_1)\}, \mathcal{A})$". Assigning the same typing Con_1^* to hole X – after mapping i_1, i_2, o_1, o_2 to d_2, d_3, d_4, d_5 resp. – allows us to derive the strong typing:

$$\begin{aligned}\mathsf{Con}_2^* \;=\; \{\; & {'d_1} : [4,6],\; i_1 : [2,3],\; i_2 : [2,3],\\ & {'d_4} : [2,3],\; {'d_5} : [2,3],\; {'d_6} : [4,6],\\ & {'d_1} = i_1 + i_2,\; {'d_1} = {'d_6},\; i_1 + i_2 = {'d_4} + {'d_5} \;\}\end{aligned}$$

for the sketch "$\textbf{conn}(\{(o_1, {'d_3}), (o_2, {'d_2})\}, X, {'\mathcal{A}})$", which formally requires invoking rules Hole (for X), Module (for ${'\mathcal{A}}$), and Connect (for connecting X and ${'\mathcal{A}}$), as well as checking that the side condition (Ct) of the latter holds.

Finally, with the typing Con_1^* for "$\textbf{loop}(\{(d_6, d_1)\}, \mathcal{A})$" and the typing Con_2^* for "$\textbf{conn}(\{(o_1, {'d_3}), (o_2, {'d_2})\}, X, {'\mathcal{A}})$", any of the 4 possible ways of placing the former in the hole of the latter does not break its strong validity. For this we invoke rule Let and check its side condition (Lt).

8. SOUNDNESS

The inference rules for typed network sketches presented in Figure 2 are sound with respect to both strong and weak versions of validity. This claim is stated formally in Theorem 3. The theorem is proven by an inductive argument for which there exist two base cases, which we state below.

Axiom 1 (Module). *If we have by the inference rule* Module *that* $\Gamma \vdash (\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathcal{C}) : C_0^*$ *then it is the case that* $V \models (\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathcal{C}) : C_0^*$.

Axiom 2 (Hole). *If we have by the inference rule* Hole *that* $\Gamma \vdash (X, \mathsf{In}, \mathsf{Out}, \{\}) : C_0^*$ *then it is the case that* $V \models (X, \mathsf{In}, \mathsf{Out}, \{\}) : C_0^*$.

Modules and holes are the basis of our inductive proof. While it is possible to construct a module $\mathcal{A}$ for which $V \not\models \mathcal{A} : C_0^*$ and holes for which $V \not\models X : C_0^*$, it is unreasonable to expect any network with such modules or holes to have a valid valuation. Thus, we assume that all modules and holes trivially satisfy our theorem.

Theorem 3 (Soundness). *If* $\Gamma \vdash \mathcal{N} : C^*$ *can be derived by the inference rules then for any* V, $V \models \mathcal{N} : C^*$.

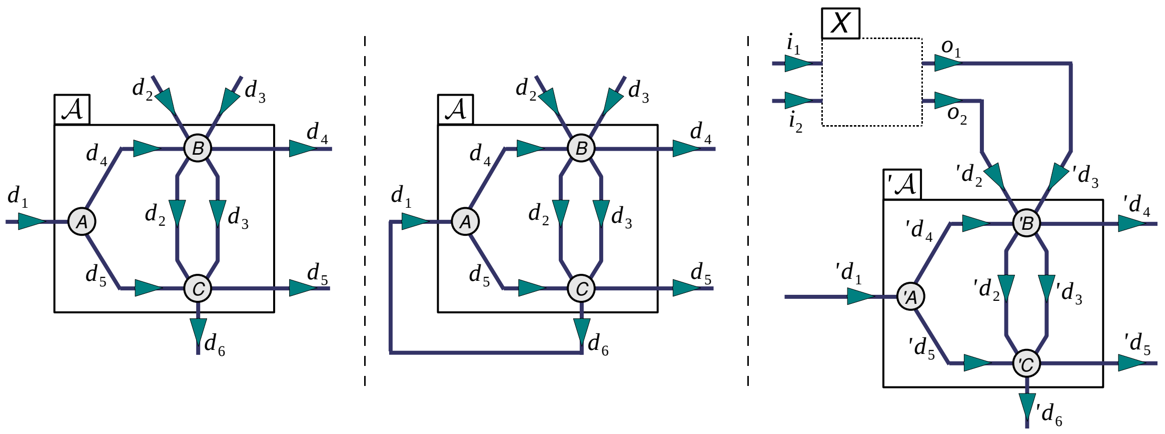

Figure 3: Graphic representation of module $\mathcal{A}$, network sketch "$\textsf{loop}(\{(d_6, d_1)\}, \mathcal{A})$", and network sketch "$\textsf{conn}(\{(o_1, 'd_3), (o_2, 'd_2)\}, X, '\mathcal{A})$". We omit "let $X \in \left\{\textsf{loop}(\{(d_6, d_1)\}, \mathcal{A})\right\}$ in $\textsf{conn}\Big(\{(o_1, 'd_3), (o_2, 'd_2)\}, X, '\mathcal{A}\Big)$ ".

Proof. The theorem holds by induction over the structure of the derivation $\Gamma \vdash \mathcal{N} : C^*$. Axioms 1 and 2 are the two base cases. The four propositions covering the four possible inductive cases can be found in the full version of this paper [6]. □

In related work [22], a significant portion of the proof has been formalized and verified using a lightweight formal reasoning and automated verification system.[9]

9. RELATED WORK

Our formalism for reasoning about constrained-flow networks was inspired by and based upon formalisms for reasoning about programs developed over the decades within the programming languages community. While our work focuses in particular on networks and constraints on flows, there is much relevant work in the community addressing the general problem of reasoning about distributed programs. However, most previously proposed systems for reasoning in general about the behavior of distributed programs (Process algebra [4], Petri nets [30], Π-calculus [28], finite-state models [25, 26, 27], and model checking [18, 19]) rely upon the retention of details about the *internals* of a system's components in assessing their interactions with one another. While this affords these systems great expressive power, that expressiveness necessarily carries with it a burden of complexity. Such an approach is inherently not modular in its analysis. In particular, the details maintained in a representation or model of a component are not easily introduced or removed. Thus, in order for a global analysis in which components are interfaced or compared to be possible, the specifications of components must be highly coordinated. Furthermore, these specifications are often wedded to particular methodologies and thus do not have the *generality* necessary to allow multiple kinds of analysis. This incompatibility between different forms of analysis makes it difficult to model and reason about how systems specified using different methodologies interact. More generally, maintaining information about internal details makes it difficult to analyze parts of a system independently and then, without reference to the internals of those parts, assess whether they can be assembled together.

Discovering and enforcing bounds on execution of program fragments is a well-established problem in computing [36], and our notion of types (*i.e.*, linear constraints) for networks can be viewed as a generalization of type systems expressing upper bounds on program execution times. Existing work on this problem includes the aiT tool (described in [32], and elsewhere), which uses control-flow analysis and abstract interpretation to provide static analysis capabilities for determining worst and best case execution time bounds. Other works, belonging to what have been called Dependent Type Systems, provide capabilities for estimating an upper bound on execution time and memory requirements via a formal type system that has been annotated with size bounds on data types. These include (but are not limited to) Static Dependent Costs [31], Sized Type Systems [20], and Sized Time Systems [24]. Many other Dependent Type Systems directly target resource bounding for the real-time embedded community (*e.g.*, the current incarnation of the Sized Time System [15], Mobile Resource Guarantees for Smart Devices [3]).

More generally, there has been a large interest in applying custom type systems to domain specific languages (which peaked in the late nineties, *e.g.*, the USENIX Conference on Domain-Specific Languages (DSL) in 1997 and 1999). Later type systems have been used to bound other resources such as expected heap space usage (*e.g.*, [17], [3]). The sup-

[9]There are other theoretical results certifying that our rules in Figures 1 and 2 work as expected. These results will be included in forthcoming reports. For example, for untyped network sketches $\mathcal{M}$, $\mathcal{N}$ and $\mathcal{P}$, which are to be connected according to the one-one maps:

$$\theta_1 \subseteq_{1\text{-}1} \textsf{Out}(\mathcal{M}) \times \textsf{In}(\mathcal{N}) \quad \text{and}$$
$$\theta_2 \subseteq_{1\text{-}1} \textsf{Out}(\textsf{conn}(\theta_1, \mathcal{M}, \mathcal{N})) \times \textsf{In}(\mathcal{P})$$

we can compute the one-one maps:

$$\theta_1' \subseteq_{1\text{-}1} \textsf{Out}(\mathcal{N}) \times \textsf{In}(\mathcal{P}) \quad \text{and}$$
$$\theta_2' \subseteq_{1\text{-}1} \textsf{Out}(\mathcal{M}) \times \textsf{In}(\textsf{conn}(\theta_1', \mathcal{N}, \mathcal{P}))$$

such that

$$\textsf{conn}(\theta_2, \textsf{conn}(\theta_1, \mathcal{M}, \mathcal{N}), \mathcal{P}) = \textsf{conn}(\theta_2', \mathcal{M}, \textsf{conn}(\theta_1', \mathcal{N}, \mathcal{P}))$$

Moreover, the valid typings (weak or strong) are exactly the same for both sides of the equation. Informally, the order in which we connect $\mathcal{M}$, $\mathcal{N}$ and $\mathcal{P}$, does not matter – whether $\mathcal{M}$ and $\mathcal{N}$ first and then appending $\mathcal{P}$, or $\mathcal{N}$ and $\mathcal{P}$ first and then prepending $\mathcal{M}$.

port for constructing, modelling, inferring, and visualizing networks and properties of network constraints provided by our work is similar to the capabilities provided by modelling and checking tools such as Alloy [21]. Unlike Alloy's system, which models constraints on sets and relations, our formalism focuses on constraints governing flows through directed graphs.

Our work intersects the body of work on Interface Theories [2, 13, 33], yet differs both in relation to motivation and the formalization of the concepts that follow from it. Like our work, Interface Theories models composition of systems by relating the interfaces of components but, unlike our work, provides a calculus of relations to describe such interfacing. Our work has grown from interest in modeling large compositions of constrained-flow networks, where not all components are known or assembled at the same time, using ideas inspired by type systems for strongly-typed programming languages, in our work, safe interfacing of components corresponds to a case of inferring or checking that a subtyping relationship holds (not unlike the observation made by [23]). We intend to carry out a careful comparison between the two approaches in forthcoming work, assessing limitations and advantages of both in the modeling and design of large safety-critical systems.

One of the essential activities our formalism aims to support is reasoning about and finding solution ranges for sets of constraints that happen to describe properties of a network. In its most general form, this is known as the *constraint satisfaction problem* [35] and is widely studied [34]. The types we have discussed in this work are linear constraints, so one variant of the constraint satisfaction problem relevant to our work involves only linear constraints. Finding solutions respecting collections of linear constraints is a classic problem that has been considered in a large variety of work over the decades. There exist many documented algorithms [11, Ch. 29] and analyses of practical considerations [14]. However, the typical approach is to consider a homogenous list of constraints of a particular class. A distinguishing feature of our formalism is that it does not treat the set of constraints as monolithic. Instead, a tradeoff is made in favor of providing users a way to manage large constraint sets through abstraction, encapsulation, and composition. Complex constraint sets can be hidden behind simpler constraints – namely, types (*i.e.*, linear constraints) that are restricted to make the analysis tractable – in exchange for a potentially more restrictive solution range. Conjunction of large constraint sets is made more tractable by employing compositional techniques.

The work in this paper extends and generalizes our earlier work in TRAFFIC (*Typed Representation and Analysis of Flows For Interoperability Checks* [5]), and complements our earlier work in CHAIN (*Canonical Homomorphic Abstraction of Infinite Network protocol compositions* [9]). CHAIN and TRAFFIC are two distinct generic frameworks for analyzing existing grids/networks, and/or configuring new ones, of local entities to satisfy desirable global properties. Relative to one particular global property, CHAIN's approach is to reduce a large space of sub-configurations of the complete grid down to a relatively small and equivalent space that is amenable to an exhaustive verification of the global property using existing model-checkers. TRAFFIC's approach uses type-theoretic notions to specify one or more desirable properties in the form of invariants, each invariant being an appropriately formulated type, that are preserved when interfacing several smaller subconfigurations to produce a larger subconfiguration. CHAIN's approach is top-down, TRAFFIC's approach is bottom-up.

While our formalism supports the specification and verification of desirable global properties and has a rigorous foundation, it remains ultimately lightweight. By lightweight we mean to contrast our work to the heavy-going formal approaches – accessible to a narrow community of experts – which are permeating much of current research on formal methods and the foundations of programming languages (such as the work on automated proof assistants [29, 16, 10, 12], or the work on polymorphic and higher-order type systems [1], or the work on calculi for distributing computing [8]). In doing so, our goal is to ensure that the constructions presented to users are the *minimum* that they might need to accomplish their task, keeping the more complicated parts of these formalisms "under the hood".

10. CONCLUSION AND FUTURE WORK

We have introduced a compositional formalism for modelling or assembling networks that supports reasoning about and analyzing constraints on flows through these networks. We have precisely defined a semantics for this formalism, and have illustrated how it can be used in specific scenarios (other examples can be found in a companion paper [7] describing NetSketch, a tool that implements this formalism). Finally, we noted that this formalism is sound with respect to its semantics in a rigorous sense (a complete formal proof of this assertion can be found in the full version of this report [6]).

In the tool that employs our formalism (NetSketch), the constraint system implemented is intended to be a proof-of-concept to enable work on typed networks (holes, types, and bounds). We intend to expand the constraint set that is supported within NetSketch to include more complex constraints. Likewise, future work involving the formalism itself could involve enriching the space of constraints. This includes both relatively straightforward extensions, such as the introduction of new relations or operators into the grammar of constraints, as well as more sophisticated ones. For instance, we have only briefly begun experimentation with making time an explicit parameter in our current framework. As a concrete example, consider the preservation of density at a fork gadget, currently defined as $d_1 = d_2 + d_3$. Time as an explicit parameter, we could describe constraints indexed with discrete time intervals (*e.g.*, $d_1(t) = d_2(t) + d_3(t)$) and can easily imagine constraints that are dependent on prior parameter values.

The equivalent of *type inference* within our formalism also deserves more attention and effort. As we indicated, there is no natural ordering of types. If no optimal constraint function is assumed, any reasonable type inference process could produce multiple, different valid types. Types can be considered optimal based on the size of their value ranges (*e.g.*, a wider or more permissive input range, and narrower or more specific output range, are preferable, in analogy with types in a strongly-typed functional language or in an object-oriented language), but even then, multiple "optimal" typings may exist. It is necessary to establish principles and algorithms by which a tool employing our formalism could assign types. Such principles or algorithms might operate by assigning weights for various valid typings.

11. REFERENCES

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

[2] L. d. Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.

[3] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, number 3362 in Lecture Notes in Computer Science, pages 1–26. Springer-Verlag, 2005.

[4] J. Baeten and W. Weijland. *Process Algebra*. Cambridge University Press, 1990.

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

[6] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: Formalism. Technical Report BUCS-TR-2009-029, CS Dept., Boston University, September 29 2009.

[7] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: Tool and Use Cases. Technical Report BUCS-TR-2009-028, CS Dept., Boston University, September 29 2009.

[8] G. Boudol. The π-calculus in direct style. In *97: 24th*, pages 228–241, 1997.

[9] A. Bradley, A. Bestavros, and A. Kfoury. Systematic Verification of Safety Properties of Arbitrary Network Protocol Compositions Using CHAIN. In *Proceedings of ICNP'03: The 11th IEEE International Conference on Network Protocols*, Atlanta, GA, November 2003.

[10] A. Ciaffaglione. *Certified reasoning on Real Numbers and Objects in Co-inductive Type Theory*. PhD thesis, Dipartimento di Matematica e Informatica Università di Udine, Italy, 2003. available as outline.

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

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

[13] L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 79–88, New York, NY, USA, 2008. ACM.

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

[15] K. Hammond, C. Ferdinand, and R. Heckmann. Towards formally verifiable resource bounds for real-time embedded systems. *SIGBED Rev.*, 3(4):27–36, 2006.

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

[17] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL '03*, pages 185–197. ACM Press, 2003.

[18] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[19] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proc. ICSE99*, pages 597–607, Los Angeles, CA, May 1999.

[20] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *ACM PoPL*, pages 410–423, 1996.

[21] D. Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[22] A. Lapets and A. Kfoury. Verification with Natural Contexts: Soundness of Safe Compositional Network Sketches. Technical Report BUCS-TR-2009-030, CS Dept., Boston University, October 16 2009.

[23] E. A. Lee and Y. Xiong. System-level types for component-based design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 237–253, London, UK, 2001. Springer-Verlag.

[24] H.-W. Loidl and K. Hammond. A sized time system for a parallel functional language. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1996.

[25] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3)(3):219–246, Sept. 1989.

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

[27] N. Lynch and F. Vaandrager. Forward and backward simulations – part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[28] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (Part I and II). *Information and Computation*, (100):1–77, 1992.

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

[30] C. A. Petri. *Communication with Automata*. PhD thesis, Univ. Bonn, 1966.

[31] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *LISP and Functional Programming*, pages 65–78, 1994.

[32] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2-3):157–179, 2000.

[33] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. On relational interfaces. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 67–76, New York, NY, USA, 2009. ACM.

[34] E. Tsang. A glimpse of constraint satisfaction. *Artif. Intell. Rev.*, 13(3):215–227, 1999.

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

[36] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.