Types as Models: Model Checking Message-Passing Programs

Sagar Chaki
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA - 15213
chaki+@cs.cmu.edu

Sriram K. Rajamani
Microsoft Research
One Microsoft Way
Redmond, WA 98052
sriram@microsoft.com

Jakob Rehof
Microsoft Research
One Microsoft Way
Redmond, WA 98052
rehof@microsoft.com

ABSTRACT

Abstraction and composition are the fundamental issues in making model checking viable for software. This paper proposes new techniques for automating abstraction and decomposition using source level type information provided by the programmer. Our system includes two novel components to achieve this end: (1) a behavioral type-and-effect system for the π-calculus, which extracts sound models as types, and (2) an assume-guarantee proof rule for carrying out compositional model checking on the types. Open simulation between CCS processes is used as both the subtyping relation in the type system and the abstraction relation for compositional model checking.

We have implemented these ideas in a tool— PIPER. PIPER exploits type signatures provided by the programmer to partition the model checking problem, and emit model checking obligations that are discharged using the SPIN model checker. We present the details on applying PIPER on two examples: (1) the SIS standard for managing trouble tickets across multiple organizations and (2) a file reader from the pipelined implementation of a web server.

1. INTRODUCTION

1.1 The problem

Distributed and asynchronous message-passing systems are notoriously hard to design and test. In this paper we focus on statically checking *behavioral* properties of such systems such as deadlock freedom, absence of race conditions, and message understood properties. In hardware and protocol design, such properties are checked by manually constructing models of the system as communicating finite state machines, and using a model checker to algorithmically explore the state space of the model. Three main issues arise when applying model checking to distributed message-passing software:

- **Expressiveness.** Unlike hardware, message-passing software tends to be more difficult to model as communicating state machines. The difficulty stems from dynamically created indirect references that are sent as messages.
- **State explosion.** State explosion is a major limitation, and severely limits the size of models that can be checked automatically. Abstraction (throwing away irrelevant information about the system and simplifying it) and composition (dividing the model checking problem into parts) are the only ways to deal with state explosion on large programs.
- **Integrating user input.** Both abstraction and composition require user assistance. User input needs to integrated with the source code tightly. Otherwise, there is no way way to check for consistency between user input and source code, and maintain this consistency as the source code evolves.

1.2 Our approach

We adopt the π-calculus [23] as a modeling language for distributed, asynchronous message-passing systems. We use a behavioral type system for extracting CCS models (which are types) from the source code, and perform model checking on the types. Our approach addresses the main issues mentioned above as follows:

- **Expressiveness.** Channel passing in the π-calculus is an expressive modeling tool for capturing several challenging idioms in communicating software systems.
- **State explosion.** Our type checker extracts abstract models from the code. An assume-guarantee principle enables us to do compositional reasoning at the level of our types.
- **Integrating user input.** User input is couched as type signatures. The type checker generates subtyping obligations to check the consistency of user annotations with the source code. Type signatures are used by our type checker to guide model extraction, and in combination with our assume-guarantee rule, type signatures support model decomposition.

POPL '02, Jan. 16-18, 2002 Portland, OR USA © 2002 ACM ISBN 1-58113-450-9/02/01...$5.00.

1.3 Contributions

This paper proposes new techniques for automating abstraction and decomposition using source level type information provided by the programmer. The technical contributions of this paper are:

- We define a new behavioral type-and-effect system for the π-calculus, based on the idea of processes-as-effects. We propose special type signatures for automating abstraction and model decomposition.
- We show that open simulation on CCS processes is a sound and natural notion of subtyping in this system. We prove a new assume-guarantee rule for open simulation in CCS, and show that it integrates into the subtyping logic of the type system.
- We present PIPER, a tool that implements our type system. PIPER uses type signatures to generate models for the program. It also generates subtyping obligations for checking the validity of abstractions provided in type signatures. PIPER has a backend to the SPIN model checker, using which we check both temporal properties on the models, and the subtyping obligations. We illustrate PIPER on two non-trivial examples.

1.4 Example

Consider the following system, where a *Sender* uses a simple message-acknowledgment protocol to send messages to a *Receiver*.

$$\begin{array}{lcl} Sender(m,a) & \triangleq & \mu\alpha.(m!.a?.\alpha + a?.Error) \\ Receiver(m,a) & \triangleq & \mu\beta.(m?.(a!.\beta + m?.Error)) \\ System_\pi & \triangleq & \left\{ \begin{array}{r} www?[m,a].Receiver(m,a) \mid \\ (\nu ma)(www![m,a].Sender(m,a)) \end{array} \right. \end{array}$$

There is a well known channel www (think about this as the URL of the *Receiver*) through which the *Sender* initiates a conversation, and establishes a pair of fresh channels m and a to carry out the conversation. Our goal in this example, is to check that the error state *Error* is never reached in the system $System_\pi$. We wish to do this without having to explore the state space of the entire system directly. We automate abstraction and decomposition guided by user annotations that are provided as type signatures.

Our type system allows the user to specify an *effect type* for the channel www of the form

$$\begin{array}{l} \widehat{Sender} \triangleq \mu\alpha.(m!.a?.\alpha) \\ \widehat{Receiver} \triangleq \mu\beta.(m?.a!.\beta) \\ www : \mathbf{ch}(m : C_m, a : C_a)\langle\!|\, \widehat{Sender} \Rightarrow \widehat{Receiver} \,|\!\rangle \end{array}$$

This type states that www is a channel on which the channels m and a are passed, which are recursively described by the types C_m and C_a. The component $\widehat{Sender} \Rightarrow \widehat{Receiver}$ is a pair of *effects* in the form of a CCS processes. The effect $\widehat{Sender}$ is a behavioral specification for *Sender* (the continuation of the receive on www) and the effect $\widehat{Receiver}$ is a behavioral specification for *Receiver* (the continuation of a send on www). With the above type signature, PIPER produces a CCS model for $System_\pi$:

$$System_{CCS} \triangleq \left\{ \begin{array}{r} www?[] \mid \\ (\nu ma)www![].(\widehat{Sender} \mid \widehat{Receiver}) \end{array} \right.$$

Further, it produces two subtyping obligations:

$$\begin{array}{c} (\eta m,a)(Sender \mid \widehat{Receiver}) \leq_{Sender} \widehat{Sender} \\ (\eta m,a)(\widehat{Sender} \mid Receiver) \leq_{Receiver} \widehat{Receiver} \end{array}$$

The η operator (defined in Section 2) is a technical device for defining which actions on m and a we must observe in order to discharge the subtyping obligations. The relation $\leq$ (defined in Section 2) denotes open simulation between CCS processes. Both these obligations are discharged using a model checker. Each obligation mentions only one component of the system $System_\pi$ on the left hand side. In general, such decomposition can lead to exponential cost savings in state exploration.

The specifications provided in the type signature of the above example make circular assumptions about each other. $\widehat{Sender}$ is a sound specification for *Sender* only when the environment of *Sender* behaves according to the specification $\widehat{Receiver}$, and $\widehat{Receiver}$ is a sound specification for *Receiver* only when the environment of *Receiver* behaves according to the specification $\widehat{Sender}$. Since behavioral abstractions are used circularly to reason about each other, the soundness of the reasoning needs to be established. Such circular proof rules are called assume-guarantee(A-G) rules, and their soundness requires an induction over time.

By soundness of our typing rules and assume-guarantee principle, we know that $System_{CCS}$ is a conservative behavioral model of system $System_\pi$. Since $System_{CCS}$ does not even contain the error state *Error*, we establish that the error state is unreachable in $System_\pi$. More generally, temporal safety properties that can be checked on $System_{CCS}$ carry over to $System_\pi$.

1.5 Paper outline

The remainder of this paper is organized as follows. In Section 2 we describe the formal framework of our paper, including our CCS semantics and π-calculus semantics. Section 3 presents our behavioral effect type system. Section 4 proves an assume-guarantee principle for CCS with open simulation. Section 5 describes how we can use our type system together with type signatures to drive assume-guarantee based model decomposition. In Section 6 we present implementation details about PIPER and illustrate PIPER on two real-life examples. Finally, Section 7 discusses related work and Section 8 concludes the paper. Due to space limitations, proofs of theorems have been left out. They can be found in our technical report [6].

2. FRAMEWORK

We define syntax and semantics for π-calculus processes [23] and CCS processes [23], and we define open simulation and process subtyping on CCS processes. Our definitions refer to Figure 14 through Figure 17, which are placed in the appendix at the end of the paper.

2.1 Syntax

Process expressions of the π-calculus are ranged over by P, Q, R etc. and are defined as follows.

(Processes)
$P ::= \mathbf{0} \mid G_1 + \ldots + G_n \mid (P_0|P_1) \mid *P \mid (\nu x : C)P$

(Guarded expressions)
$G ::= x!^t[\vec{y}].P \mid x?^t[\vec{y} : \vec{C}].P$

In processes, x, y, z range over *channel names*. We write $\vec{x}$ etc. for vectors of variables. Our π-calculus processes are typed, with type expressions ranged over by C. Type expressions are defined in Section 3. The process $x!^t[\vec{z}].P$ sends the channels $\vec{z}$ on channel x and continues as P. The process $x?^t[\vec{y} : \vec{C}].P$ receives channels on x and binds them to $\vec{y}$ in the continuation P. The process $*P$ replicates parallel copies of P. Following [19], we use *tags*, ranged over by t, to label individual send and receive operations. As in [19], tags are useful for stating properties of a process, and they serve additional, technical purposes in Section 4.

CCS process expressions are ranged over by Γ, Δ etc. and are defined as follows.

(Processes)
$$\Gamma ::= \mathbf{0} \mid \alpha \mid \gamma_1 + \ldots + \gamma_n \mid (\Gamma_0|\Gamma_1) \mid \mu\alpha.\Gamma \mid (\nu x)\Gamma$$

(Guarded expressions)
$$\gamma ::= x!^t.\Gamma \mid x?^t.\Gamma$$

We write $*\Gamma$ as an abbreviation for $\mu\alpha.(\Gamma \mid \alpha)$.

2.2 Semantics

Structural congruence for π-calculus processes is defined in Figure 14. Reduction semantics of π-calculus processes is defined in Figure 15. A substitution of names for names is denoted σ. Substitutions are defined on CCS and π-calculus process expressions in the natural way. A substitution with finite support, written $\{\vec{x} \mapsto \vec{y}\}$, denotes the simultaneous substitution of $\vec{y}$ for $\vec{x}$ (the vectors of names, $\vec{x}$ and $\vec{y}$, are assumed to be of equal length), acting as the identity on names not in $\vec{x}$. Applications of such substitutions are written in postfix form, as in $\Gamma\{\vec{x} \mapsto \vec{y}\}$.

Structural congruence for CCS processes is defined in Figure 16. Reductions on CCS processes is defined in Figure 17. In addition to reactions, we introduce commitments in rules O-COMM and I-COMM for CCS. Commitments enable us to do compositional reasoning. In addition to the usual rules for the restriction operator ν we have rules ETA1 and ETA2 for a special operator η, which we add in order to state our assume-guarantee rule. An expression of the form $(\eta x)\Gamma$ allows us to observe reactions (but not commmitments) taking place on the channel x.

Let Act be the set of all actions of the form x^{t_1,t_2}, $x!^t$, $x?^t$,τ^{t_1,t_2} or ϵ, and let a range over elements of Act. If $\Gamma \xrightarrow{a_1} \Gamma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \Gamma'$, then the sequence $a_1a_2\ldots a_n$ is called a *trace* of Γ. Traces are ranged over by ω. We consider traces *modulo* ϵ, that is ϵ may be tacitly inserted or deleted from traces. For a trace ω, let ω^τ denote the trace that arises from ω by eliding all τ actions. Also, let ω° denote the sequence that arises from ω by replacing all actions of the form x^{t_1,t_2}, $x!^t$ or $x?^t$ with x. For a trace ω, we define the *norm* of ω, denoted $|\omega|$, to be the sequence $(\omega^\tau)^\circ$. We write $\omega =_{\mathsf{N}} \omega'$ as an abbreviation for $|\omega| = |\omega'|$. We write $\Gamma \stackrel{a}{\Longrightarrow} \Gamma'$ if and only if $\Gamma \xrightarrow{\omega} \Gamma'$ with $\omega =_{\mathsf{N}} a$.

2.3 Simulation and subtyping

Subtyping is the principle of model abstraction in our type system to be presented in Section 3. We fix subtyping on CCS processes to be open simulation.

DEFINITION 2.1. *(Open Simulation) A binary relation $\mathcal{R}$ on CCS processes is called an* open simulation relation *iff $\Gamma \mathrel{\mathcal{R}} \Delta$ implies that, for every σ, a and Γ',*

$$\sigma(\Gamma) \xrightarrow{a} \Gamma' \Rightarrow \exists\Delta'.\ \sigma(\Delta) \stackrel{a}{\Longrightarrow} \Delta' \text{ and } \Gamma' \mathrel{\mathcal{R}} \Delta'$$

□

Our notion of open simulation is identical to the notion of weak open simulation studied by Sangiorgi [30]. We drop the qualifier "weak" for brevity. We note that any open simulation relation $\mathcal{R}$ is closed under substitutions, that is, $\Gamma \mathrel{\mathcal{R}} \Delta$ implies $\sigma(\Gamma) \mathrel{\mathcal{R}} \sigma(\Delta)$, for all σ. We define our notion of subsumption of CCS processes, called process subtyping, as follows.

DEFINITION 2.2. *(Process Subtyping, $\leq$) We define the* process subtyping *relation on CCS processes, denoted $\leq$, to be the largest open simulation on CCS processes.* □

3. BEHAVIORAL TYPE SYSTEM

We seek a way for programmers to state properties of message passing programs, in such a way that the programmer can control abstraction and model checking in a compositional way. Our solution is based on a new types and effects system, the core of which we present in this section (see Figure 1). We will extend the system further in Section 5 (see Figure 3), and Section 6 contains several examples of its use. Our type system is inspired by the systems of Gordon and Jeffrey [13] and of Igarashi and Kobayashi [19]. We defer more detailed comparisons with previous work to Section 7.

3.1 Typing rules

The type structure in our system is based on the idea of *channel types* with *latent behavioral effects*. Type expressions are ranged over by C and are defined as follows:

(Effect types) $\quad C ::= \mathbf{ch}(\vec{x} : \vec{C})\langle\!|\,\Gamma\,|\!\rangle$

We write $\mathbf{ch}$ as abbreviation for $\mathbf{ch}()\langle\!|\ |\!\rangle$. A π-calculus channel x may be assigned a type $\mathbf{ch}(\vec{y} : \vec{C})\langle\!|\,\Gamma\,|\!\rangle$, expressing that x denotes a channel on which other channels can be passed, which are recursively described by $\vec{C}$. The names $\vec{y}$ have binding occurrences in the type shown, and they are used to range over the channels passed on x. In the component $\langle\!|\,\Gamma\,|\!\rangle$, Γ is a CCS process. We regard this process as a *latent effect*, describing the actions that a process can or may take on channels (ranged over by $\vec{y}$) passed along x, and possibly other channels. Technically, our types are *dependent*, since effects (Γ) are parameterized on names ($\vec{y}$).

Our core type system, called $\mathcal{E}$, is defined in Figure 1. The rules shown in Figure 2 are auxiliary rules, defining well formed types, type environments and lookup. Type environments E are lists of bindings of types to channel names, of the form $E ::= \vec{x} : \vec{C}$. The domain of an environment E is $\mathsf{dom}(E) = \{x \mid x : C \in E\}$.

The type system defines derivable judgments of the form $E \vdash \Gamma \rhd P$, where E is an environment, P is a π-calculus process, and Γ is a CCS model of P, which soundly approximates the behavior of P. The relation between Γ and P is made precise in Theorem 3.1 below.

Key rules are the input and output rules, [T-INP] and [T-OUT], and the subsumption rule, [SUB]. In rule [T-INP], the process Γ_0 describes the effects of the input continuation on the received channels $\vec{y}$ (and, possibly other channels) on x. These effects are *captured* in the latent effect on the type of x. At a corresponding output on x, the latent effect will be *realized*, via rule [T-OUT], as a concurrent process $\Gamma_0\{\vec{y} \mapsto \vec{z}\}$, which is instantiated with the passed values $\vec{z}$ and running in parallel with the output continuation.

In rule [SUB], the subtyping relation $\leq$ is open simulation on CCS processes (Definition 2.1). Rule [SUB] allows us to perform model *abstraction*, based on simulation.

As an example typing, consider again our $System_\pi$ from Section 1. With Γ given by

$$\Gamma \triangleq (\nu www)(www? \mid (\nu ma)(www!.Sender(m,a) \mid Receiver(m,a)))$$

one can derive the judgment

$$\emptyset \vdash \Gamma \rhd (\nu www : \mathbf{ch}(m : \mathbf{ch}, a : \mathbf{ch})\langle\!\langle Sender(m,a) \rangle\!\rangle) System_\pi$$

Section 5 and Section 6 contain more examples of how the type system works.

3.2 Basic Properties

The following theorem contains the soundness result for our system. The proof is given in our technical report [6]. The proof makes essential use of the simulation property of subtyping. In comparison to [19], our notion of subtyping-as-simulation allows us to prove soundness with respect to standard CCS reaction semantics for our models Γ (see Section 7 for further discussion).

THEOREM 3.1 (SUBJECT REDUCTION). *If $E \vdash \Gamma \rhd P$ and $P \xrightarrow{a} P'$, then there exists Γ' such that $\Gamma \xRightarrow{a} \Gamma'$ and $E \vdash \Gamma' \rhd P'$.*

As an immediate corollary of Theorem 3.1, we can soundly check safety properties on models Γ derivable in the type system: if $E \vdash \Gamma \rhd P$ and $\Gamma \models \Box\psi$, then $P \models \Box\psi$.

Our next result shows that our choice of subtyping as open simulation is natural for the notion of type soundness contained in Theorem 3.1. To see this, let us consider variations of our type system that arise by reinterpreting the subtyping relation, in principle by taking any binary relation $\mathcal{R}$ on CCS terms to be the subtyping relation (instead of $\leq$) in the typing rules. Writing $\Gamma' \mathrel{\mathcal{R}} \Gamma$, we mean that Γ' is a supertype of Γ. Let us denote the type system constructed this way as $\mathcal{E}_\mathcal{R}$. One then has $E \vdash \Gamma \rhd P$ and $\Gamma' \mathrel{\mathcal{R}} \Gamma$ together imply $E \vdash \Gamma' \rhd P$, by rule [SUB], in system $\mathcal{E}_\mathcal{R}$. We can prove the following theorem, which shows that our notion of subtyping as open simulation is the most powerful notion of subsumption possible for our system $\mathcal{E}$ relative to our notion of soundness (Theorem 3.1) and the requirement that typings be closed under substitution. The proof can be found in [6].

THEOREM 3.2. *Let $\mathcal{R}$ be a binary relation on CCS processes such that the system $\mathcal{E}_\mathcal{R}$ satisfies the following properties:*

1. *$\mathcal{E}_\mathcal{R}$ satisfies the subject reduction property.*
2. *$\mathcal{E}_\mathcal{R}$ satisfies that the relation $\rhd$ is closed under substitutions.*

Then $\mathcal{R} \subseteq \leq$.

One can define syntactic abstractions on CCS models, which are easy to compute. An example is the abstraction function $\uparrow_S$ of [19]. We can show that a modified version is admissible in our effect system, because it leads to an open simulation relation contained in $\leq$. However, we note that our definition is different, in that our abstraction depends on tags and not channel names. See [6] for details.

4. ASSUME-GUARANTEE PRINCIPLE

In this section we prove an assume-guarantee principle for CCS with respect to open simulation. This principle can be used to reason about CCS processes compositionally, and by exploiting it in our type system we open the door to compositional specification and model checking at the level of types. Section 5 describes extensions to our type system to support assume-guarantee reasoning.

Given two CCS processes Θ and Ω, suppose we want to check if $\Theta \leq \Omega$. Suppose further that $\Theta = (\nu\vec{x})(\Gamma \mid \Delta)$ is a composition of two processes Γ and Δ that interact over a set of restricted channels $\vec{x}$, and that the abstract process $\Omega = (\nu\vec{x})(\Gamma' \mid \Delta')$ is structurally similar to the concrete process Θ. We desire to check if $\Theta \leq \Omega$ without exploring the entire state space of Θ directly. Theorem 4.3 provides a way to do this.

We wish to consider a process Δ' as an abstraction of Δ in the context $(\nu\vec{x})(\Gamma \mid \Delta)$ (and similarly, an abstraction Γ' of Γ in that context). The essence of Theorem 4.3 is that, on the restricted channels $\vec{x}$, we can simplify the abstraction Δ' in ways that would not otherwise be possible. Intuitively, with respect to the channels in $\vec{x}$, the abstraction Δ' only needs to simulate the interactions that Δ can perform in the context of Γ. This is so, because we know that no further (unknown) environment can interfere on the channels in $\vec{x}$, due to the name restriction $\nu\vec{x}$. The expression $\eta\vec{x}$ singles out channels on which reactions can be observed, and is a technical means to reason about these requirements for Δ'.

To prepare for Theorem 4.3, we need a few definitions. Our abstraction Δ' must match reactions of Δ on $\vec{x}$ by commitments on $\vec{x}$, because we will consider Δ' without any operating environment. To state this requirement, we need to be able to split traces. For an action a occurring in a trace of $(\eta\vec{x})(\Gamma \mid \Delta)$ we define the projections $(a)_\Gamma$ and $(a)_\Delta$ as follows, by cases over the form of a:

$$
\begin{array}{lcl}
(x^{t_1,t_2})_\Gamma & = & \left\{\begin{array}{ll} x^{t_1,t_2} & \text{if } t_1 \in \mathsf{T}(\Gamma) \text{ and } t_2 \in \mathsf{T}(\Gamma) \\ x!^{t_1} & \text{if } t_1 \in \mathsf{T}(\Gamma) \text{ and } t_2 \notin \mathsf{T}(\Gamma) \\ x?^{t_2} & \text{if } t_1 \notin \mathsf{T}(\Gamma) \text{ and } t_2 \in \mathsf{T}(\Gamma) \\ \epsilon & \text{if } t_1 \notin \mathsf{T}(\Gamma) \text{ and } t_2 \notin \mathsf{T}(\Gamma) \end{array}\right. \\
(\tau^{t_1,t_2})_\Gamma & = & \left\{\begin{array}{ll} \tau^{t_1,t_2} & \text{if } t \in \mathsf{T}(\Gamma) \text{ and } t_2 \in \mathsf{T}(\Gamma) \\ \epsilon & \text{if } t_1 \notin \mathsf{T}(\Gamma) \text{ or } t_2 \notin \mathsf{T}(\Gamma) \end{array}\right. \\
(x!^t)_\Gamma & = & \left\{\begin{array}{ll} x!^t & \text{if } t \in \mathsf{T}(\Gamma) \\ \epsilon & \text{if } t \notin \mathsf{T}(\Gamma) \end{array}\right. \\
(x?^t)_\Gamma & = & \left\{\begin{array}{ll} x?^t & \text{if } t \in \mathsf{T}(\Gamma) \\ \epsilon & \text{if } t \notin \mathsf{T}(\Gamma) \end{array}\right. \\
(\epsilon)_\Gamma & = & \epsilon
\end{array}
$$

The projection $(a)_\Delta$ is defined analogously. If ω is a trace of $(\eta\vec{x})(\Gamma \mid \Delta)$, we define the *projection of ω onto* Γ, denoted ω_Γ, to be given by $(\omega_\Gamma)_{[i]} = (\omega_{[i]})_\Gamma$, for $i = 1 \ldots n$, where $\omega_{[i]}$ is the i'th element of ω.

We also need to be able to combine traces. For this purpose, define the partial function $\oplus$ on $\mathsf{Act} \times \mathsf{Act}$ by setting $x!^t \oplus x?^{t'} = x^{t,t'}$ and $a \oplus \epsilon = \epsilon \oplus a = a$ for all $a \in \mathsf{Act}$. We lift to traces of equal length by defining $(\omega_1 \oplus \omega_2)_{[i]} = (\omega_1)_{[i]} \oplus (\omega_2)_{[i]}$, $i = 1 \ldots n$. For a set of channel names $\vec{x}$, we define the relation $\vec{x} \vdash a \sim a'$ to hold

$$E \vdash \mathbf{0} \rhd \mathbf{0} \quad [\text{T-NULL}]$$

$$\frac{E, x : C \vdash \Gamma \rhd P}{E \vdash (\nu x)\Gamma \rhd (\nu x : C)P} \quad [\text{T-NEW}]$$

$$\frac{\begin{array}{c} E \vdash x : \mathbf{ch}(\vec{y} : \vec{C})\langle\!|\, \Gamma_0 \,|\!\rangle \\ E, \vec{y} : \vec{C} \vdash \Gamma_0 \mid \Gamma_1 \rhd P \\ \vec{y} \cap \mathsf{fn}(\Gamma_1) = \emptyset \end{array}}{E \vdash x?^t.\Gamma_1 \rhd x?^t[\vec{y} : \vec{C}].P} \quad [\text{T-INP}]$$

$$\frac{\begin{array}{c} E \vdash x : \mathbf{ch}(\vec{y} : \vec{C})\langle\!|\, \Gamma_0 \,|\!\rangle \\ E \vdash \Gamma \rhd P \\ E \vdash \langle \vec{z} \rangle : \langle \vec{y} : \vec{C} \rangle \end{array}}{E \vdash x!^t.(\Gamma \mid \Gamma_0\{\vec{y} \mapsto \vec{z}\}) \rhd x!^t[\vec{z}].P} \quad [\text{T-OUT}]$$

$$\frac{E \vdash \Gamma_0 \rhd P_0 \qquad E \vdash \Gamma_1 \rhd P_1}{E \vdash \Gamma_0 \mid \Gamma_1 \rhd P_0 \mid P_1} \quad [\text{T-PAR}]$$

$$\frac{E \vdash \Gamma \rhd P}{E \vdash *\Gamma \rhd *P} \quad [\text{T-REP}]$$

$$\frac{E \vdash \gamma_i \rhd G_i \quad (i = 1 \ldots n)}{E \vdash \sum_i \gamma_i \rhd \sum_i G_i} \quad [\text{T-SUM}]$$

$$\frac{E \vdash \Gamma \rhd P \qquad \Gamma \leq \Gamma'}{E \vdash \Gamma' \rhd P} \quad [\text{T-SUB}]$$

Figure 1: Behavioral type and effect system ($\mathcal{E}$)

$$\emptyset \vdash \diamond \quad [\text{ENV1}]$$

$$\frac{E; x \vdash C \qquad x \notin \mathsf{dom}(E)}{E, x : C \vdash \diamond} \quad [\text{ENV2}]$$

$$\frac{\begin{array}{c} E, \vec{x} : \vec{C} \vdash \diamond \qquad y \notin \mathsf{dom}(E) \\ \mathsf{fn}(\Gamma) \subseteq \mathsf{dom}(E) \cup \vec{x} \cup \{y\} \end{array}}{E; y \vdash \mathbf{ch}(\vec{x} : \vec{C})\langle\!|\, \Gamma \,|\!\rangle} \quad [\text{ENV3}]$$

$$\frac{E, x : C, E' \vdash \diamond}{E, x : C, E' \vdash x : C} \quad [\text{ENV4}]$$

$$E \vdash \langle\rangle : \langle\rangle \quad [\text{DEC1}]$$

$$\frac{E \vdash \langle \vec{x} \rangle : \langle \vec{y} : \vec{C} \rangle \qquad E \vdash x : C\{\vec{y}, y \mapsto \vec{x}, x\}}{E \vdash \langle \vec{x}, x \rangle : \langle \vec{y} : \vec{C}, y : C \rangle} \quad [\text{DEC2}]$$

Figure 2: Environment rules

$$\frac{\begin{array}{c} E \vdash x : \mathbf{ch}(\vec{y} : \vec{C})\langle\!|\, \Gamma' \Rightarrow \Delta' \,|\!\rangle \\ E, \vec{y} : \vec{C} \vdash \Delta \mid \Gamma \rhd P \\ \vec{y} \cap \mathsf{fn}(\Gamma) = \emptyset \\ (\eta \vec{y})(\Gamma' \mid \Delta) \leq_\Delta \Delta' \end{array}}{E \vdash x?^t.\Gamma \rhd x?^t[\vec{y} : \vec{C}].P} (*) \quad [\text{T-INP-I}]$$

$$\frac{\begin{array}{c} E \vdash x : \mathbf{ch}(\vec{y} : \vec{C})\langle\!|\, \Gamma' \Rightarrow \Delta' \,|\!\rangle \\ E \vdash \Gamma \rhd P \\ E, \vec{z} : \vec{C}' \vdash \langle \vec{z} \rangle : \langle \vec{y} : \vec{C} \rangle \\ (\eta \vec{z})(\Gamma \mid \Delta'_{\vec{z}}) \leq_\Gamma \Gamma'_{\vec{z}} \end{array}}{E \vdash (\nu \vec{z})(x!^t.(\Gamma'_{\vec{z}} \mid \Delta'_{\vec{z}})) \rhd \overline{x}!^t[\vec{z} : \vec{C}'].P} (\dagger) \quad [\text{T-OUT-I}]$$

Side-conditions:
For some division of (I_1, I_2) of $(\vec{y}, \vec{z})$ depending on x:
$(*)$ For all $i \in I_1$, y_i is non-blocking for Δ in $(\eta \vec{y})(\Gamma' \mid \Delta)$.
$(\dagger)$ For all $j \in I_2$, z_j is non-blocking for Γ in $(\eta \vec{z})(\Gamma \mid \Delta'_{\vec{z}})$.

Notation:
In rule [T-OUT-I], $\Gamma'_{\vec{z}}$ is a shorthand for $\Gamma'\{\vec{y} \mapsto \vec{z}\}$, and $\Delta'_{\vec{z}}$ is a shorthand for $\Delta'\{\vec{y} \mapsto \vec{z}\}$.

Figure 3: Additional typing rules for symmetric channels

if and only if $a \oplus a'$ is defined and satisfying, for all $x \in \vec{x}$, that $(a = x!^t \vee a = x?^t) \Rightarrow a' \neq \epsilon$, and $(a' = x!^t \vee a' = x?^t) \Rightarrow a \neq \epsilon$. We lift this relation to traces of equal length by setting $\vec{x} \vdash \omega_1 \sim \omega_2$ if and only if $\vec{x} \vdash (\omega_1)_{[i]} \sim (\omega_2)_{[i]}$, $i = 1 \ldots n$.

Finally, we generalize the notion of open simulation under projections, as follows.

Definition 4.1. *(Open Simulation) For any CCS process Θ, let $\leq_\Theta$ be the largest relation on CCS processes satisfying that $\Gamma \leq_\Theta \Delta$ implies, for every σ, a and Γ',*

$$\sigma(\Gamma) \xrightarrow{a} \Gamma' \Rightarrow \exists \Delta'.\ \sigma(\Delta) \overset{(a)_\Theta}{\Longrightarrow} \Delta' \text{ and } \Gamma' \leq_\Theta \Delta'$$

If $\Gamma \leq_\Theta \Delta$, then we say that Γ is open simulated *by Δ* with respect to *Θ.* □

Notice that $\Gamma \leq \Delta$ if and only if $\Gamma \leq_\Gamma \Delta$. Hence, we can use $\Gamma \leq \Delta$ as an abbreviation for $\Gamma \leq_\Gamma \Delta$.

Definition 4.2. *(Nonblocking channel) Let x be a channel name in $\vec{x}$. We say that x is a* non-blocking *channel of process Γ in the process $(\eta\vec{x})(\Gamma \mid \Delta)$ if and only if whenever the following conditions hold:*

1. $\Gamma \xrightarrow{\omega_1} \Gamma'$
2. $\Delta \xrightarrow{\omega_2} \Delta'$
3. $\vec{x} \vdash \omega_1 \sim \omega_2$
4. $\Gamma' \equiv (\ldots + a^{t_1}.\Gamma'' + \ldots)$

then we have

$$\Delta' \xrightarrow{\tau^*} (\ldots + \vec{a}^{t_2}.\Delta'' + \ldots)$$

where τ^* is some sequence of τ actions, $a = x?$ and $\vec{a} = x!$, or $a = x!$ and $\vec{a} = x?$ for some x. □

We are now ready to state our assume-guarantee theorem. The proof can be found in [6].

Theorem 4.3 (Assume-Guarantee). *For any processes Γ, Δ, Γ', Δ' suppose*

A1. $(\eta\vec{x})(\Gamma \mid \Delta') \leq_\Gamma \Gamma'$

A2. $(\eta\vec{x})(\Gamma' \mid \Delta) \leq_\Delta \Delta'$

A3. for all x in $\vec{x}$, either x is non-blocking for $\sigma(\Gamma)$ in $(\eta\vec{x})(\Gamma \mid \Delta')$ for all substitutions σ, or x is non-blocking for $\sigma(\Delta)$ in $(\eta\vec{x})(\Gamma' \mid \Delta)$ for all substitutions σ.

Then we have

$$(\eta\vec{x})(\Gamma \mid \Delta) \leq (\eta\vec{x})(\Gamma' \mid \Delta')$$

Theorem 4.3 entails that the following rule [AG] is sound, given the appropriate non-blocking side-conditions:

$$\frac{\begin{array}{c}(\eta\vec{x})(\Gamma \mid \Delta') \leq_\Gamma \Gamma' \\ (\eta\vec{x})(\Gamma' \mid \Delta) \leq_\Delta \Delta'\end{array}}{(\nu\vec{x})(\Gamma \mid \Delta) \leq (\nu\vec{x})(\Gamma' \mid \Delta')}\ [\text{AG}]$$

To see that the non-blocking side conditions are necessary in Theorem 4.3, consider that our assume-guarantee result would be unsound if we left them out. Let $\Gamma \triangleq x!^t$ and $\Delta \triangleq x?^{t'}$. Disregarding the non-blocking conditions, the premises $(\eta x)(\Gamma \mid \mathbf{0}) \leq_\Gamma \mathbf{0}$ and $(\eta x)(\mathbf{0} \mid \Delta) \leq_\Delta \mathbf{0}$ would be satisfied. But the conclusion, that $(\eta x)(\Gamma \mid \Delta) \leq (\eta x)(\mathbf{0} \mid \mathbf{0})$, is not true.

5. COMPOSITIONAL ABSTRACTION

In this section we extend our core type system of Figure 1 to incorporate the assume-guarantee principle from Section 4. In Section 4 we noted that it is only useful to invoke rule [AG] at name restrictions. If, during type checking, we have arrived at a typing judgment of the form $E \vdash (\nu\vec{x})(\Gamma \mid \Delta) \triangleright P$, then rule [AG] can be readily applied to prove a given abstraction $(\nu\vec{x})(\Gamma \mid \Delta) \leq (\nu\vec{x})(\Gamma' \mid \Delta')$. This rule already allows us to do compositional model checking, provided that we specify an abstraction for a process of the form shown above. However, it would be desirable if abstractions could be automatically built up compositionally from local abstractions specified in type signatures for each channel. Section 5.2 describes our proposal for doing this.

5.1 Bounded Signatures

It is not obvious how to incorporate rule [AG] into the type system in a smooth way. To illustrate some of the problems, let us first consider a very general solution, where we introduce *bounded* type signatures of the form

$$x : \mathbf{ch}(\vec{y}\ :\ \vec{C})\ \langle\!|\, \ell : \Gamma_0 \leq \Delta_0 \,|\!\rangle$$

The interpretation of this is that Δ_0 is an abstraction of Γ_0, but only in the context signified by the label ℓ. Discharging the proof obligation $\Gamma_0 \leq \Delta_0$ is therefore deferred to the context designated ℓ, which is a restriction $(\nu^\ell\vec{y})P$. Typing judgments are extended to the form $S; E \vdash \Gamma \triangleright P$, where S collects deferred constraints $\Gamma_0 \leq \Delta_0$ that are in scope. At the designated name restriction $(\nu^\ell\vec{y})P$, the type checker can emit the proof obligation $(\nu\vec{y})\Gamma_S \leq (\nu\vec{y})\Gamma^S$, where Γ_S is the inferred CCS model for P using the "concrete" effects Γ_0, and Γ^S is the model inferred using the "abstract" effects Δ_0. Rule [AG] may be used to prove the obligation if Γ is a parallel composition.

The bounded signature scheme has a number of drawbacks. First, the signature must supply both a concrete effect, Γ, and an abstract one, Δ. Second, in some cases the scheme does not achieve much locality. Our earlier example $System_\pi$ illustrates this problem. For this system, the abstraction $Sender \mid Receiver \leq \widehat{Sender} \mid \widehat{Receiver}$ must be specified as a single bounded signature in the latent effect for channel w. The problems discussed above lead us to search for restricted yet useful communication idioms, which can be handled with simpler and more local specifications. The remainder of this section focuses on this issue.

5.2 Symmetric Channels and Signatures

In this section we observe that channels used according to the symmetric fragment of the π-calculus, also called πI, allows us to apply assume-guarantee based decomposition in an especially simple and appealing way. We present an extension to our type system, which automates a strong form of compositional abstraction for symmetric communication idioms.

The symmetric π-calculus πI was studied by Sangiorgi [29]. It is defined by restricting output prefixes in the π-calculus to *bound output*, here written $\overline{x}![\vec{z}].P$, and with the meaning

$$\overline{x}!^t[\vec{z}].P \triangleq (\nu\vec{z})(x!^t[\vec{z}].P)$$

and reaction rule

$$(\dots + x?^t[\vec{y}].P + \dots) \mid (\dots + \overline{x}!^{t'}[\vec{z}].Q + \dots) \xrightarrow{x^{t,t'}}$$
$$(\nu\vec{z})(P\{\vec{y} \mapsto \vec{z}\} \mid Q)$$

Since our type system has the ability to discipline communication on a per-channel basis, we do not have to restrict ourselves to a subcalculus to take advantage of special idioms. Instead, we introduce the notion of a *symmetric channel*, as follows.

DEFINITION 5.1. *(Symmetric channel) A channel x is symmetric in a process P, if every output on x in P is bound, of the form $\overline{x}![\vec{z} : \vec{C}].P$.* □

In order to take advantage of the bound output idiom on symmetric channels, we will introduce a new binary type constructor, $\Rightarrow$, on CCS processes, to form type expressions $\Gamma \Rightarrow \Delta$.

To exploit bound output, we add the rules shown in Figure 3 to those of Figure 1. We refer to the new system as $\mathcal{E}I$. The side-conditions in Figure 3 use the concept of a *division* of name vectors: given vectors $\vec{y}$ and $\vec{z}$ both of length n and $I_j \subseteq \{1, \dots, n\}$ for $j = 1, 2$, we say that (I_1, I_2) is a division of $(\vec{y}, \vec{z})$ if $I_1 \cup I_2 = \{1, \dots, n\}$.

The distinctive feature of system $\mathcal{E}I$ is the incorporation of a new kind of type signatures, called *symmetric signatures*, of the form

$$x : \mathbf{ch}(\vec{y} : \vec{C})\langle\!\lvert\, \Gamma' \Rightarrow \Delta' \,\rvert\!\rangle$$

In this signature, the effect $\Gamma' \Rightarrow \Delta'$ expresses mutual, cyclic assumptions of senders and receivers on channel x. With this signature, the channel x is required to be symmetric. Senders expect receivers to behave according to Δ', and receivers expect senders to behave according to Γ'. Typing a receiver, we must establish one half of the premises of rule [AG], as expressed in the premise

$$(\eta\vec{y})(\Gamma' \mid \Delta) \leq_\Delta \Delta'$$

of rule [T-INP-I]. Establishing the premise means that the model Δ from the receiver satisfies the specification Δ', *assuming* that receivers satisfy Γ'. Conversely, typing a sender, we must establish the other half of the premises of rule [AG] with respect to its model Γ and the specifications $\Gamma'_{\vec{z}}$ and $\Delta'_{\vec{z}}$, via rule [T-OUT]. In this rule, the notation $\Gamma'_{\vec{z}}$ is a shorthand for $\Gamma'\{\vec{y} \mapsto \vec{z}\}$ (and similarly for $\Delta'_{\vec{z}}$) to express that the specifications Γ' and Δ' are renamed in terms of the names $\vec{z}$ sent on x. Section 5.3 and Section 6 contain examples of how symmetric signatures can be used.

Soundness for the system $\mathcal{E}I$ can be established by changing the proof of Theorem 3.1 slightly, using the rule [AG]. The changes needed are shown in [6].

An important property of the rules [T-INP-I] and [T-OUT-I] is that the premises of rule [AG] are distributed and discharged *locally*: given the symmetric signature

$$x : \mathbf{ch}(\vec{y} : \vec{C})\langle\!\lvert\, \Gamma' \Rightarrow \Delta' \,\rvert\!\rangle$$

and a division (I_1, I_2), we can discharge the obligation

$$(\eta\vec{y})(\Gamma' \mid \Delta) \leq_\Delta \Delta'$$

without knowing the actual sender process, and we can can discharge the obligation $(\eta\vec{z})(\Gamma \mid \Delta'_{\vec{z}}) \leq_\Gamma \Gamma'_{\vec{z}}$ without knowing the actual receiver process (Δ models the receiver, and Γ the sender). Rule [AG] implies that one then has

$$(\nu\vec{z})(\Delta_{\vec{z}} \mid \Gamma) \leq (\nu\vec{z})(\Delta'_{\vec{z}} \mid \Gamma'_{\vec{z}})$$

This means that abstractions of senders and receivers on symmetric channels can be built up compositionally, and they can be verified by local checks, in terms of the specification $\Gamma' \Rightarrow \Delta'$, without the need to have the entire system assembled prior to checking.

5.3 Combinators

We give examples of how we can use simple language extensions, called combinators, in the vein of Section 5.2, to express special idioms of module composition that can be supported by the type checker. The combinators will be used in our case studies in Section 6.

Our first pair of combinators, called **accept** and **request**, are just intended to make the symmetric idiom more explicit and readable, by using keywords for both sends and receives on a symmetric channel. The typing rules for these combinators are directly derived from the rules [T-INP-I] and [T-OUT-I] from Section 5.2.

$$\begin{array}{lcl} \mathbf{accept}\ a(\vec{x} : \vec{C})\ \ \mathbf{in}\ P & \triangleq & a?[\vec{x} : \vec{C}].P \\ \mathbf{request}\ a(\vec{z} : \vec{C}')\ \ \mathbf{in}\ Q & \triangleq & (\nu\vec{z} : \vec{C}')(a![\vec{z}].Q) \end{array}$$

Using our rules [T-INP-I] and [T-OUT-I] we can now handle our example $System_\pi$ by assuming the type signature

$$w : \mathbf{ch}(m : \mathbf{ch}, a : \mathbf{ch})\langle\!\lvert\, \widehat{Sender} \Rightarrow \widehat{Receiver} \,\rvert\!\rangle$$

as promised in Section 1. Using our combinators and closing the system, it becomes

$$\begin{array}{l} (\nu w : \mathbf{ch}(m : \mathbf{ch}, a : \mathbf{ch})\langle\!\lvert\, \widehat{Sender} \Rightarrow \widehat{Receiver} \,\rvert\!\rangle) \\ \quad \mathbf{accept}\ w(m : \mathbf{ch}, a : \mathbf{ch})\ \ \mathbf{in}\ Receiver \mid \\ \quad \mathbf{request}\ w(m : \mathbf{ch}, a : \mathbf{ch})\ \ \mathbf{in}\ Sender \end{array}$$

The symmetric paradigm does not exhaust all cases where strong locality of abstraction is possible. We can still handle many such cases by simple language extensions that identify the relevant idiom. For example, we can define combinators **group** and **join**, which allow a finite number of processes to form a group communicating on a set of shared channels. For more details see [6].

6. PIPER

We have implemented a typechecker, called PIPER for an extended version of our core type system. The extensions have been driven by the usability and effectiveness of the language in describing non-trivial real-life systems. Specifically, we allow named π and CCS processes, tail-recursive process calls, *if-then-else* selectors, and pure data to be passed on CCS channels.

The input to PIPER is a set of π processes annotated with type signatures, and a set of X-free LTL formulas that we want to verify of the system. The propositions in the formulas correspond to process names. A proposition is true in a particular state of the system iff the control of some component is within the corresponding process. The goal of PIPER is to check the LTL formulae on the π system. PIPER does this by first constructing the CCS processes which are behavioral types of the π system, and then checking the temporal properties on the types. PIPER also generates subtyping obligations that relate type signatures to components of the

π system. The typechecking fails if a subtyping obligation cannot be discharged.

PIPER uses the model-checker SPIN [17] to both verify the temporal properties and discharge the subtyping obligations (recall that these involve checking open simulations between CCS processes). We have implemented a backend for translating CCS processes to equivalent PROMELA [16] programs, which can then be verified exhaustively using SPIN. Open simulation is checked by proving certain temporal properties on the system obtained by composing the two CCS processes in a particular way. We have implemented another backend that can do this composition and output the result as a PROMELA program. SPIN is then used for verifying the property. Since we use SPIN, which is a finite state model checker, this approach works when the CCS process have finite number of states. Even for systems with infinite number of states, we could abstract the processes to a ν-free fragment and check properties automatically using techniques from infinite state model checking [7]. We plan to consider this possibility in the future. In the remainder of this section, we present two examples that showcase the practical significance of the results presented in the preceding sections, and the effectiveness of PIPER in analyzing real-life systems.

6.1 SIS Protocol

The *Service Incident Exchange Standard* (SIS) [9] is a data, transaction and state model designed to enable service incident tracking systems to share service incident data and facilitate resolutions. The standard envisages interaction between service requesters and providers through distributed maintenance of an FSM (shown in Figure 4). Each service request that is processed has an associated copy of the FSM that tracks the status of the request as it gets modified by various service incidents. By separately maintaining a consistent view of the FSM, the requester and the provider can keep track of the state of the request at each point of time.

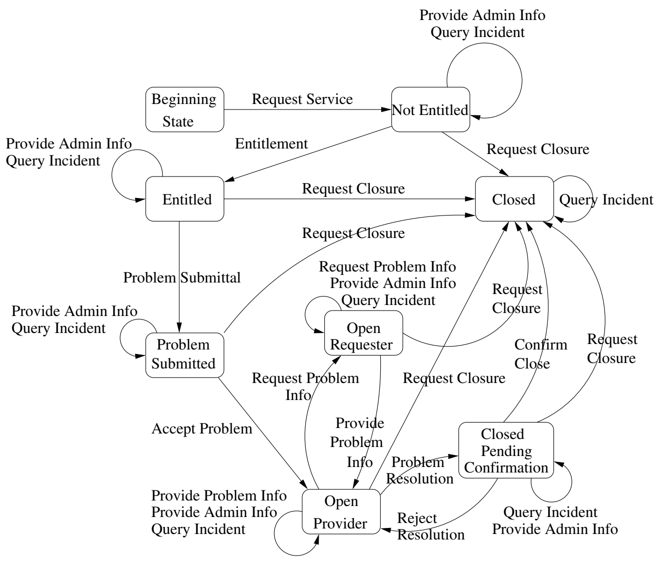

Figure 4: SIS finite state machine

Both principals maintain a copy of a virtual machine (VM) that represents the FSM. Either principal can try change the state of his VM by initiating a transaction. Not all transactions can be initiated by everyone, e.g. *Confirm Close* can be initiated only by a service requester, *Entitlement* can be initiated only by a service provider, and *Query Incident* can be initiated by either. If the transaction is valid for the initiator and the current state, the transition is taken. However the change in one VM must be immediately reflected in the other. It is the onus of the principals to ensure that the states of the two VMs are consistent at all times and behavior beyond those allowed by FSM is never observed. For example a critical requirement is that once a service request is terminated, it is never reopened. The transitions in Figure 4 are labeled with transactions that can cause them.

Figure 5 depicts only the high-level structure of a π process implementing this protocol. The requester and provider are modeled by *Req* and *Prov* respectively. Each is again composed of two components, the client (represented by *RClient* and *PClient*) which transmits valid transaction requests, and the VM which processes these requests. Each VM is modeled by a set of eight mutually recursive processes (concrete processes), one for each state of the FSM. For example, *RBegSt* models the requester's VM (RVM) in the *Beginning State* and *ROpPro* models the RVM in the *Open Provider* state. Their counterparts for the provider's VM (PVM) are *PBegSt* and *POpPro*. Channels x and y are used by the clients to communicate with their VMs.

$$
\begin{aligned}
&\mathit{Req}(r:\mathbf{ch}(\mathbf{num}), p:\mathbf{ch}(\mathbf{num})) \triangleq (\nu\ x:\mathbf{ch}(\mathbf{num}))\\
&\qquad \mathit{RClient}\langle x\rangle \mid \mathit{RBegSt}\langle x,r,p\rangle\\[1ex]
&\mathit{RBegSt}(x:\mathbf{ch}(\mathbf{num}), r:\mathbf{ch}(\mathbf{num}), p:\mathbf{ch}(\mathbf{num})) \triangleq\\
&\quad x?[n_1].\ \mathtt{if}\ (n_1 = \mathtt{SRV\_REQ})\\
&\qquad \mathtt{then}\ r![\mathtt{NOT\_ENT}].\mathit{RNotEnt}\langle x,r,p\rangle\\
&\qquad \mathtt{else}\ \mathit{RBegSt}\langle x,r,p\rangle\\
&\quad +\ p?[n_2].\mathit{Error}\langle\rangle\\
&\qquad\qquad \vdots\\
&\mathit{ROpPro}(x:\mathbf{ch}(\mathbf{num}), r:\mathbf{ch}(\mathbf{num}), p:\mathbf{ch}(\mathbf{num})) \triangleq \ldots\\[1ex]
&\mathit{Prov}(r:\mathbf{ch}(\mathbf{num}), p:\mathbf{ch}(\mathbf{num})) \triangleq (\nu\ y:\mathbf{ch}(\mathbf{num}))\\
&\qquad \mathit{PClient}\langle y\rangle \mid \mathit{PBegSt}\langle y,r,p\rangle\\[1ex]
&\mathit{PBegSt}(y:\mathbf{ch}(\mathbf{num}), r:\mathbf{ch}(\mathbf{num}), p:\mathbf{ch}(\mathbf{num})) \triangleq \ldots\\
&\qquad\qquad \vdots\\
&\mathit{POpPro}(y:\mathbf{ch}(\mathbf{num}), r:\mathbf{ch}(\mathbf{num}), p:\mathbf{ch}(\mathbf{num})) \triangleq \ldots\\[1ex]
&\mathit{main}() \triangleq (\nu\ w:\mathbf{ch}(r:\mathbf{ch}(\mathbf{num}),\\
&\quad p:\mathbf{ch}(\mathbf{num})) \langle\!|\ \mathit{R\hat{B}egSt}\langle r,p\rangle \bowtie \mathit{P\hat{B}egSt}\langle r,p\rangle\ |\!\rangle)\\
&\mathbf{request}\ w(r:\mathbf{ch}(\mathbf{num}), p:\mathbf{ch}(\mathbf{num}))\ \ \mathbf{in}\ \mathit{Req}\langle r,p\rangle \mid\\
&\mathbf{accept}\ w(r':\mathbf{ch}(\mathbf{num}), p':\mathbf{ch}(\mathbf{num}))\ \ \mathbf{in}\ \mathit{Prov}\langle r',p'\rangle
\end{aligned}
$$

Figure 5: Some π processes for the SIS protocol

At each step the RVM non-deterministically decides either to accept a transaction request from *RClient* over channel x, or a request for a state transition from PVM over channel p. In the first case, it checks for the validity of the transaction in the current state. On success it issues a request for a state change to PVM over channel r and then changes its own state. In the second case it checks if the state change requested by PVM is legal in the current state. If so, it changes its state, otherwise it goes to the *Error* state. The situation is symmetric on the provider's side.

State transition in each VM is effected by calling the appropriate process. Consider *RBegSt* described in Figure 5. If RVM decides to receive a transaction n_1 from *RClient*, it checks if the transaction is a *Request Service*. If so it requests PVM to go to *Not Entitled* state and itself goes to that state. Otherwise it remains in the *Beginning State*. If RVM decides to accept a state change request n_2 from PVM, it goes to *Error* state since, in the beginning state, PVM cannot legally cause any state change.

The abstraction for the system is supplied as a set of abstract CCS processes (shown in Figure 6), one for each concrete process. Each abstract process abstracts the behavior of the corresponding concrete process, e.g. $\widehat{RBegSt}$ abstracts *RBegSt*. The abstraction simplifies the system at two levels. First, the interaction with the client is completely elided e.g. there is no abstract process for *RClient*. Second, the abstract processes do not contain error handlers, since they make assumptions about the environment (note the absence of *Error* in $\widehat{RBegSt}$ and its simplicity compared to *RBegSt*). These abstractions are given to PIPER in the symmetric signature for channel w in process *main* shown in Figure 5.

$$\widehat{RBegSt}(r : \mathbf{ch}(\mathbf{num}), p : \mathbf{ch}(\mathbf{num})) \triangleq \\ r![\mathtt{NOT\_ENT}].\widehat{RNotEnt}\langle p, r\rangle$$

$$\vdots$$

$$\widehat{ROpPro}(r : \mathbf{ch}(\mathbf{num}), p : \mathbf{ch}(\mathbf{num})) \triangleq \ldots$$

$$\widehat{PBegSt}(r : \mathbf{ch}(\mathbf{num}), p : \mathbf{ch}(\mathbf{num})) \triangleq \ldots$$

$$\vdots$$

$$\widehat{POpPro}(r : \mathbf{ch}(\mathbf{num}), p : \mathbf{ch}(\mathbf{num})) \triangleq \ldots$$

Figure 6: Abstract processes for the SIS protocol

For each of the π processes, PIPER produced a CCS process type with identical name. The type produced for the *main* process is shown in Figure 7. In order to verify any property of *main*, the model checker has to deal only with the abstract processes, which are much simpler, and have smaller state spaces. In addition PIPER generated and discharged the subtyping obligations of Figure 8. Finally PIPER verified several critical properties of the system, some of which are tabulated in Figure 9. Each row depicts a property with the corresponding LTL formula(s). For a more detailed account of this case study see [6].

$$main() \triangleq (\nu\ w : \mathbf{ch})(\nu\ r : \mathbf{ch}(\mathbf{num}))(\nu\ p : \mathbf{ch}(\mathbf{num})) \\ w!.(\widehat{RBegSt}\langle r, p\rangle \mid \widehat{PBegSt}\langle r, p\rangle) \mid w?$$

Figure 7: Main type for the SIS protocol

$$\left(\begin{array}{c}(\eta\ r : \mathbf{ch}(\mathbf{num}))(\eta\ p : \mathbf{ch}(\mathbf{num})) \\ Req\langle r, p\rangle \mid \widehat{PBegSt}\langle r, p\rangle\end{array}\right) \leq_{Req} \widehat{RBegSt}\langle r, p\rangle$$

$$\left(\begin{array}{c}(\eta\ r : \mathbf{ch}(\mathbf{num}))(\eta\ p : \mathbf{ch}(\mathbf{num})) \\ \widehat{RBegSt}\langle r, p\rangle \mid Prov\langle r, p\rangle\end{array}\right) \leq_{Prov} \widehat{PBegSt}\langle r, p\rangle$$

Figure 8: Subtyping obligations for the SIS protocol

6.2 Webserver

We model a file reader from the pipelined implementation of a webserver. This example originated from the *Staged-Server* [21] project at MSR. The file is logically divided into

Property	LTL formula
The two VMs never reach two distinct states of the FSM simultaneously - these formulas are all true	$\Box\neg(x \wedge y)$, $(x, y) \in \hat{Q}$
All the states of both VMs are reachable - these formulas are all false	$\Box\neg x$, $x \in \hat{P} \cup \hat{R}$
Once a request is closed neither party can reopen it - these formulas are all true	$\Box\neg(\widehat{RClo} \wedge \Diamond r)$, $r \in \hat{R} \setminus \{\widehat{RClo}\}$
Once a request is entitled it cannot be unentitled	$\Box\neg(\widehat{REnt} \wedge \Diamond \widehat{RNotEnt})$

Figure 9: SIS properties verified by Piper. $\hat{P}$ and $\hat{R}$ are abstract states of the provider's and requester's VM. $\hat{Q}$ contains pairs of states from $\hat{P}$ and $\hat{R}$ which correspond to distinct states of the FSM.

a number of blocks. The task of reading each block from the disk and transmitting it over the network is handled by separate processes. The blocks can be read in parallel from the disk but must be transmitted in proper sequence. The objective is to mitigate the disk I/O bottleneck. Figure 10 describes the π process for the system.

$$Sender(p : \mathbf{ch}(c : \mathbf{ch}, d : \mathbf{ch}(\mathbf{num}); n : \mathbf{num}) \\ \langle\!|\ \widehat{Sender}\langle c, d; n\rangle\ |\!\rangle) \triangleq \\ p?[c', d'; n'].\mathtt{if}\ (n' = \mathtt{MAX})\ \mathtt{then}\ c'?.d'![n'] \\ \mathtt{else}(\nu\ c'' : \mathbf{ch})p![c'', d'; (n'+1)].c'?.d'![n'].c''!$$

$$Receiver(c : \mathbf{ch}(\mathbf{num}); n : \mathbf{num}) \triangleq c?[n']. \\ \mathtt{if}\ (n \neq n')\ \mathtt{then}\ Error\langle\rangle\ \mathtt{else}\ Receiver\langle c'; (n+1)\rangle$$

$$main() \triangleq (\nu\ a : \mathbf{ch}(\mathbf{num}))(\nu\ b : \mathbf{ch}(c : \mathbf{ch}, d : \mathbf{ch}(\mathbf{num}); \\ n : \mathbf{num})\langle\!|\ \widehat{Sender}\langle c, d; n\rangle\ |\!\rangle)(\nu\ c : \mathbf{ch}) \\ b![c, a; 0].c! \mid *\ Sender\langle b\rangle \mid Receiver\langle a; 0\rangle$$

Figure 10: π process for the webserver

Sender models the process that reads and transmits one block of the file. First it reads three values from channel p - c',d' and n'. c' is the channel on which it must wait for a signal before it can transmit the index n' of its block on channel d'. Note that for the purpose of verifying that the blocks are transmitted in sequence, it suffices to reason only in terms of their indices.

If n' refers to the last block to be transmitted, *Sender* waits on c' and then transmits n'. Otherwise it creates a fresh channel c'' and transmits c'', d' and $(n'+1)$ on p. This data is received by another copy of *Sender* (c'' will act as c' for this copy). Then it waits on c', transmits n' and signals the copy which is now waiting on c''. *Receiver* models the process that receives the file blocks. *main* models the entire system consisting of a process that starts off the initial *Sender*, infinite copies of *Sender*s and a *Receiver*.

The proof that the blocks are transmitted in sequence is non-trivial owing to the channel hiding in *Sender* and the way these fresh channels are used by one copy of *Sender* to signal another. The abstract behavior of the sender is modeled by the two CCS processes described in Figure 11. The types produced are described in Figure 12.

In addition the typechecker emitted the typing obligation

$$\hat{Sender}(c:\mathbf{ch}, d:\mathbf{ch}(\mathbf{num}); n:\mathbf{num}) \triangleq \\ c?.\hat{SendFrom}\langle d; n\rangle$$

$$\hat{SendFrom}(c:\mathbf{ch}(\mathbf{num}); n:\mathbf{num}) \triangleq \\ \texttt{if } (n = \texttt{MAX}) \texttt{ then } c![n] \\ \texttt{else } c![n].\hat{SendFrom}\langle c; (n+1)\rangle$$

Figure 11: Abstract user-defined type for the webserver

$$Sender(p:\mathbf{ch}) \triangleq p?.\texttt{if } (\bot) \texttt{ then } 0 \texttt{ else } 0$$

$$Receiver(c:\mathbf{ch}(\mathbf{num}); n:\mathbf{num}) \triangleq c?[n']. \\ \texttt{if } (n \neq n') \texttt{ then } Error\langle\rangle \texttt{ else } Receiver\langle c; (n+1)\rangle$$

$$main() \triangleq (\nu\ a:\mathbf{ch}(\mathbf{num}))(\nu\ b:\mathbf{ch})(\nu\ c:\mathbf{ch}) \\ b!.(c!|\hat{Sender}\langle c, a; 0\rangle) \mid *\ Sender\langle b\rangle \mid Receiver\langle a; 0\rangle$$

Figure 12: CCS types for the webserver

$$\left(\begin{array}{c} c?.\texttt{if } (n = \texttt{MAX}) \texttt{ then } d![n] \\ \texttt{else } (\nu\ b:\mathbf{ch}) \\ (d![n].b! \mid \hat{Sender}\langle b, d; (n+1)\rangle) \end{array}\right) \leq \hat{Sender}\langle c, d; n\rangle$$

Figure 13: Subtyping obligation for webserver

shown in Figure 13. For a fixed value of MAX, this obligation can be discharged by PIPER. We were able to do this for MAX=20. In general, this obligation can be discharged by induction on n. It is also obvious that the CCS processes comprising of the types transmit the file blocks in sequence.

7. RELATED WORK

Since the work of Nielson and Nielson [25], behavioral type systems have received increasing attention. Recent work includes [13, 19, 5, 26, 28, 31, 11, 12, 18]. In distinction to traditional type systems, which focus on data abstraction, behavioral type systems focus on abstracting control structures of concurrent programs. For behavioral type systems, it is therefore natural to attempt integrating techniques from model checking into the type checker. Model checking has a long history, for which we refer the reader to [8].

The core rules of our type system are inspired by two previous lines of work. One is the effect type system for the π-calculus by Gordon and Jeffrey [13] and the other is the generic type system for the π-calculus by Igarashi and Kobayashi [19]. A novel feature of our system in comparison to [13] is that our effects are "computationally active", whereas effects in [13] are not. In comparison to [19], we use channel types with latent effects as processes and symmetric type signatures to leverage programmer input, and automate model extraction and compositional reasoning. On the downside, our system is not able to type as many processes as the system in [19], because a channel can only have one type in our system.

Whereas [19] uses the ν-free fragment of CCS in their models, our system uses full CCS. In particular, we exploit hiding via name restriction at the CCS level to support compositional model checking via assume-guarantee principles. Together with our previous work [27], our system appears to be the first attempt to exploit these ideas in the setting of behavioral type systems.

Our notion of subtyping-as-simulation allows us to work with standard CCS reaction semantics for our CCS models. In the system of [19], CCS semantics is dependent upon subtyping, which is an abstract relation defined by an axiom system.

The present paper continues our work [27] on defining behavioral module systems for message-passing programming languages. In comparison to [27], the present paper has a different focus— automating abstraction and decomposition using source level type information provided by the programmer. Our new type-and-effect system incorporating an assume-guarantee rule for open simulation (rather than trace containment) serves this purpose. Finally, we have implemented a new tool, PIPER, based on the theory presented in this paper, and we have applied it to model check a number of systems and protocols.

Assume-guarantee rules that allow apparently circular assumptions about operating contexts can be traced back to [24, 1, 2]. Recent work has used such techniques to model check large hardware circuits [3, 22, 10, 14]. All these rules are based on trace containment, require a nonblocking assumption on the process calculus, and are not directly applicable to CCS. In this paper we prove a new assume-guarantee rule for CCS with respect to open simulation, in order to integrate it with the subtyping logic of the type system. The proof technique for establishing the soundness of our assume-guarantee rule builds on ideas from our earlier work in [27] and [15]. An alternative approach to reason with context-sensitive abstractions is given in [20]. We are aware of two model checkers [4, 22] that provide tool support for assume-guarantee reasoning. Both these systems do not support dynamic channel creation and channel passing, which are important features in distributed software. In addition, the novel aspects of this paper are the integration of assume-guarantee reasoning into the subtyping logic of a type system, and the use of type signatures to guide the decomposition.

Model checking full CCS is undecidable in general. Decidable fragments include the finite control fragment and the ν-free fragment [7]. We use the SPIN model checker [17] to discharge finite state model checking obligations.

Open simulation was studied by Sangiorgi [30], only here we consider simulation with respect to observations on reductions. Our symmetric signatures take advantage of the properties of πI [29] on a per-channel basis, without restriction to a subcalculus.

8. CONCLUSION

We have proposed new techniques for automating abstraction and decomposition using source level type information provided by the programmer. Type checking is used to extract a model from the program, and model checking is used on the extracted model. User annotations in the form of type signatures guide both model extraction and compositional model checking. Our type-and-effect system and assume-guarantee rule are both novel and required nontrivial soundness proofs. We have implemented a tool PIPER that interfaces with the SPIN model checker to discharge the model checking obligations. We have presented the experience of using PIPER on two real-life examples

APPENDIX

This appendix contains Figure 14 through Figure 17, which are placed at the end of the paper. These definitions are discussed in Section 2.

Acknowledgements

We thank Greg Meridith for discussions on behavioral type systems and message-passing programs and for pointing us to the SIS standard. We thank Jim Larus for providing us with the web server example and explaining its intricacies. We thank the members of the programming languages group in Microsoft Research-Cambridge for several constructive comments on this work.

A. REFERENCES

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[3] R. Alur and T. A. Henzinger. Reactive modules. In *LICS 96: Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[4] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA : Modularity in model checking. In *CAV 98: Computer Aided Verification*, LNCS, pages 521–525. Springer-Verlag, 1998.

[5] T. Amtoft, F.Nielson, and H.R. Nielson. *Type and Effect Systems. Behaviours for Concurrency*. Imperial College Press, 1999.

[6] S. Chaki, S.K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. Technical report, Microsoft Research, 2001. Available at `research.microsoft.com/behave`.

[7] S. Christensen, Y. Hirshfeld, and F. Moller. Decidable subsets of CCS. *The Computer Journal*, 37(4):233–242, 1994.

[8] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] Customer Support Consortium (CSC) and Desktop Management Task Force (DMTF). Service incident standard (sis) specification, version 1.1. Technical report, DMTF. Available at `www.dmtf.org`.

[10] A.T. Eiriksson. The formal design of 1M-gate ASICs. In *FMCAD 98: Formal Methods in Computer-Aided Design*, LNCS 1522, pages 49–63. Springer-Verlag, 1998.

[11] C. Flanagan and M. Abadi. Types for safe locking. In *ESOP 99: European Symposium on Programming*, LNCS 1576, pages 91–108. Springer-Verlag, 1999.

[12] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI 00: Programming Language Design and Implementation*, pages 219–232. ACM, 2000.

[13] A. Gordon and A. Jeffrey. Typing corresondence assertions for communication protocols. In *MFPS: Mathematical Foundations of Programming Semantics*, pages 99–120. BRICS Notes Series NS-01-2, 2001.

[14] T. A. Henzinger, X. Liu, S. Qadeer, and S. K. Rajamani. Formal specification and verification of a dataflow processor array. In *ICCAD 99:Computer-Aided Design*, pages 494–499. IEEE Computer Society Press, 1999.

[15] T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In *FMCAD 98: Formal Methods in Computer-aided Design*, LNCS 1522, pages 421–432. Springer-Verlag, 1998.

[16] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[17] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[18] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP 98*. Springer, 1998.

[19] A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. In *POPL 01: Principles of Programming Languages*, pages 128–141. ACM, 2001.

[20] Kim G. Larsen and Robin Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99:80–108, 1992.

[21] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. Technical Report MSR-TR-2001-39, Microsoft Research, 2001.

[22] K. L. McMillan. A compositional rule for hardware design refinement. In *CAV 97: Computer-Aided Verification*, LNCS 1254, pages 24–35. Springer-Verlag, 1997.

[23] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[24] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[25] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *POPL 94: Principles of Programming Languages*, pages 84–97. ACM, 1994.

[26] F. Puntigam and C. Peter. Changeable interfaces and promised messages for concurrent components. In *SAC 99: Symposium on Applied Computing*, pages 141–145. ACM, 1999.

[27] S.K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *SAS 01: Static Analysis*, LNCS 2126, pages 375–394. Springer-Verlag, 2001.

[28] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR 00: Concurrency Theory*, LNCS 1877, pages 474–488. Springer-Verlag, 2000.

[29] D. Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(2), 1996.

[30] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33, 1996.

[31] N. Yoshida. Graph types for monadic mobile processes. In *FSTTCS: Software Technology and Theoretical Computer Science*, LNCS 1180, pages 371–387. Springer-Verlag, 1996.

$$P \equiv P \mid \mathbf{0} \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$(\nu x : C)(P \mid Q) \equiv P \mid (\nu x : C)Q \quad (\text{if } x \notin \mathsf{fn}(P))$$

$$(\nu x : C)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x : C)(\nu y : C')P \equiv (\nu y : C')(\nu x : C)P$$

$$*P \equiv P \mid *P$$

Structural congruence is the congruence relation on π-calculus terms induced by the following axioms, together with renaming of bound variables and reordering of terms in a summation.

Figure 14: Structural congruence for π

$$(\ldots + x!^{t}[\vec{z}].P + \ldots) \mid (\ldots + x?^{t'}[\vec{y} : \vec{C}].Q + \ldots) \xrightarrow{x^{t,t'}} P \mid Q\{\vec{y} \mapsto \vec{z}\} \quad [\textsc{react}]$$

$$\frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \quad [\textsc{par}] \qquad \frac{P \equiv P' \quad P' \xrightarrow{\ell} Q' \quad Q \equiv Q'}{P \xrightarrow{\ell} Q} \quad [\textsc{cong}]$$

$$\frac{P \xrightarrow{x^{t,t'}} P'}{(\nu x : C)P \xrightarrow{\tau^{t,t'}} (\nu x : C)P'} \quad [\textsc{new1}] \qquad \frac{P \xrightarrow{\ell} P' \quad x \notin \ell}{(\nu x : C)P \xrightarrow{\ell} (\nu x : C)P'} \quad [\textsc{new2}]$$

Figure 15: π-calculus reaction semantics

$$\Gamma \equiv \Gamma \mid \mathbf{0} \qquad \Gamma_0 \mid \Gamma_1 \equiv \Gamma_1 \mid \Gamma_0 \qquad \Gamma_0 \mid (\Gamma_1 \mid \Gamma_2) \equiv (\Gamma_0 \mid \Gamma_1) \mid \Gamma_2$$

$$(\nu x)(\Gamma_0 \mid \Gamma_1) \equiv \Gamma_0 \mid (\nu x)\Gamma_1 \quad (\text{if } x \notin \mathsf{fn}(\Gamma_0))$$

$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x)(\nu y)\Gamma \equiv (\nu y)(\nu x)\Gamma$$

$$\mu\alpha.\Gamma \equiv \Gamma\{\alpha \mapsto \mu\alpha.\Gamma\}$$

Structural congruence is the congruence relation on CCS terms induced by the following axioms, together with renaming of bound variables and reordering of terms in a summation.

Figure 16: Structural congruence for CCS

$$\Gamma \xrightarrow{\epsilon} \Gamma \qquad [\textsc{eps}]$$

$$(\ldots + x!^t.\Gamma_0 + \ldots) \mid (\ldots + x?^{t'}.\Gamma_1 + \ldots) \xrightarrow{x^{t,t'}} \Gamma_0 \mid \Gamma_1 \quad [\textsc{red}]$$

$$(\ldots + x!^t.\Gamma + \ldots) \xrightarrow{x!^t} \Gamma \qquad [\text{O-}\textsc{comm}]$$

$$(\ldots + x?^t.\Gamma + \ldots) \xrightarrow{x?^t} \Gamma \qquad [\text{I-}\textsc{comm}]$$

$$\frac{\Gamma \xrightarrow{\ell} \Gamma'}{\Gamma \mid \Gamma_0 \xrightarrow{\ell} \Gamma' \mid \Gamma_0} \quad [\textsc{par}] \qquad \frac{\Gamma_0 \equiv \Gamma_0' \quad \Gamma_0' \xrightarrow{\ell} \Gamma_1' \quad \Gamma_1 \equiv \Gamma_1'}{\Gamma_0 \xrightarrow{\ell} \Gamma_1} \quad [\textsc{cong}]$$

$$\frac{\Gamma \xrightarrow{x^{t,t'}} \Gamma'}{(\nu x)\Gamma \xrightarrow{\tau^{t,t'}} (\nu x)\Gamma'} \quad [\textsc{new}1] \qquad \frac{\Gamma \xrightarrow{\ell} \Gamma' \quad x \notin \ell}{(\nu x)\Gamma \xrightarrow{\ell} (\nu x)\Gamma'} \quad [\textsc{new}2]$$

Eta rules

$$\frac{\Gamma \xrightarrow{x^{t,t'}} \Gamma'}{(\eta x)\Gamma \xrightarrow{x^{t,t'}} (\eta x)\Gamma'} \quad [\textsc{eta}1] \qquad \frac{\Gamma \xrightarrow{\ell} \Gamma' \quad x \notin \ell}{(\eta x)\Gamma \xrightarrow{\ell} (\eta x)\Gamma'} \quad [\textsc{eta}2]$$

In the rules above, ℓ ranges over actions of the form $x!^t$,$x?^t$, x^{t_1,t_2} or τ^{t_1,t_2}.

Figure 17: CCS semantics