A Behavioral Module System for the Pi-Calculus

Sriram K. Rajamani and Jakob Rehof

Microsoft Research
{sriram,rehof}@microsoft.com

Abstract. Distributed message-passing based asynchronous systems are becoming increasingly important. Such systems are notoriously hard to design and test. A promising approach to help programmers design such programs is to provide a *behavioral type system* that checks for behavioral properties such as deadlock freedom using a combination of type inference and model checking. The fundamental challenge in making a behavioral type system work for realistic concurrent programs is state explosion. This paper develops the theory to design a *behavioral module system* that permits decomposing the type checking problem, saving exponential cost in the analysis. Unlike module systems for sequential programming languages, a behavioral specification for a module typically assumes that the module operates in an appropriate concurrent context. We identify assume-guarantee reasoning as a fundamental principle in designing such a module system.
Concretely, we propose a behavioral module system for π-calculus programs. Types are CCS processes that correctly approximate the behavior of programs, and by applying model checking techniques to process types one can check many interesting program properties, including deadlock-freedom and communication progress. We show that modularity can be achieved in our type system by applying circular assume-guarantee reasoning principles whose soundness requires an induction over time. We state and prove an assume-guarantee rule for CCS. Our module system integrates this assume-guarantee rule into our behavioral type system.

1 Introduction

Several computing systems are built today in a distributed wide-area setting, using an asynchronous message-passing programming model. These programs are notoriously hard to design and test, due to inherent difficulties in dealing with concurrency. Better programming languages and programming tools for building such programs are becoming increasingly important.

In hardware and protocol design, there has been success in modeling different agents as communicating finite state machines, and using model checking to explore the interactions between the agents. However, agents in concurrent software tend to have more complicated communication structure than their counterparts in hardware. Indirect references and dynamic creation of new objects play a prominent role in interactions between software agents. For instance, one agent can create a new object and send the object's reference to a second agent. Following this, both agents can read or change the object's contents. Such

P. Cousot (Ed.): SAS 2001, LNCS 2126, pp. 375–394, 2001.

interactions are typically hard to model using communicating finite state machines. The π-calculus provides a simple way to model such interactions. The combination of fresh name generation and channel passing allows faithful modeling of several complicated communication patterns between software agents. However, in spite of its simple semantics, it is hard to automatically analyze π-calculus programs for checking behavioral properties. Recently, there has been considerable interest in designing so called *behavioral type systems* for statically checking important behavioral properties such as deadlock freedom and communication progress for π-calculus programs. Behavioral type systems use type inference to extract behavioral abstractions of the program, called *behavioral types*, and use model checking to explore the state space of these behavioral types. The fundamental obstacle in making a behavioral type system scale is the exponential state space explosion in model checking. The only hope for dealing with state explosion on realistic programs is to partition the type checking and model checking problems to operate on pieces of the program, thereby saving exponential amount of analysis time. We develop the theory required to design a *behavioral module system*, which makes such partitioning possible.

Our work is inspired by the behavioral type systems proposed by Igarashi and Kobayashi [11]. Here, types are CCS-like processes that correctly approximate the behavior of π-calculus programs, and types are inferred from programs. A model checker is used as a subroutine inside the typechecker for checking interesting program properties, including deadlock-freedom and communication progress. In this paper, we propose to incorporate *assume-guarantee* reasoning [17,1,3] to enable modular type checking in such a system. Assume-guarantee reasoning allows the programmer to state behavioral abstractions of a module that hold only in contexts where the module will actually be used.

Since our types are CCS processes, we need an assume-guarantee rule that works for CCS. All known assume-guarantee rules require the process calculus to have a nonblocking semantics. Since CCS processes can block, previous assume-guarantee results are not directly applicable. This paper has three technical contributions:

- We state and prove an assume-guarantee rule for CCS.
- We propose a behavioral type system for π-calculus in which types are CCS processes. Our type system is a variant of Igarashi and Kobayashi's type system, and it includes name restriction in the process types.
- We show that name restriction in CCS allows for a natural integration of our assume-guarantee rule into the behavioral type system.

There are significant technical hurdles in designing a behavioral module system for a concurrent programming language. Module systems for sequential programming languages such as ML allow the user to specify abstractions of modules using type signatures. Module systems tactfully combine analysis and user annotation to partition type checking. A type signature of a module in ML is typically independent of the sequential context where the module is used. However, it is often impossible to state useful behavioral abstractions of a module that hold in all concurrent contexts. This phenomenon is well known in the specification and verification of reactive systems [17,1,3]. Thus, we need to allow the user to state

$S = \mu\alpha.(x?.m!.a?.\alpha + a?.(m!)^*)$
$U_s = \mu\gamma.(x!.\gamma)$
$Sender = \nu x.(S \mid U_s)$
$\hat{Sender} = \mu\alpha.(m!.a?.\alpha)$

$R = \mu\beta.(m?.(y!.a!.\beta + m?.(a!)^*))$
$U_r = \mu\delta.(y?.\delta)$
$Receiver = \nu y.(R \mid U_r)$
$\hat{Receiver} = \mu\beta.(m?.a!.\beta)$

$System = \nu m, a.(Sender \mid Receiver)$

Fig. 1. A Sender and Receiver in CCS.

behavioral abstractions of a module that hold only in contexts where the module will actually be used. The resulting module system needs to reason about a module using behavioral abstractions of its environments. For instance, if we have two concurrent modules P and Q with behavioral specifications P' and Q', then we assume Q' as the environment for establishing that P' is a correct abstraction of P. Similarly, we assume P' as the environment for establishing that Q' is a correct abstraction of Q. 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. We identify assume-guarantee reasoning as a fundamental principle in designing a behavioral module system.

The remainder of the paper is organized as follows. Section 2 contains two examples illustrating various aspects of our module system. In Section 3 we state and prove an assume-guarantee rule for CCS. In Section 4 we propose a behavioral module system for π-calculus. In Section 5 we discuss related work, and Section 6 concludes the paper.

2 Examples

We show two examples, one to illustrate the assume-guarantee rule and one to illustrate our type system. We follow the syntax for CCS and π-calculus from [16].

Figure 1 shows a *Sender* process sending messages to a *Receiver* process. The *Sender* and *Receiver* communicate through a message channel m and an acknowledgement channel a, that are known apriori to both processes. *Sender* comprises of process S that does the actual communication, and a local user process U_s which is consulted before every message transmission. *Receiver* comprises of process R that does the actual communication, and a local user process U_r which is consulted after every message reception. Suppose we want to check a safety property of *System* such as deadlock freedom, specified by a temporal-logic formula ψ. One way to do this is to explore the state space of *System* using a model checker. In order to alleviate state explosion, it is useful to write

abstractions of the components of the system, and run the model checker on each component separately. If the user writes abstract specifications $\hat{Sender}$ and $\hat{Receiver}$ for the sender and receiver respectively, one could attempt using the following compositional proof rule to avoid exploring the state space of *System*.

$$\frac{\begin{array}{l} Sender \subseteq \hat{Sender} \\ Receiver \subseteq \hat{Receiver} \\ (\eta m, a)(\hat{Sender} \mid \hat{Receiver}) \models \psi \end{array}}{(\eta m, a)(Sender \mid Receiver) \models \psi} \quad \text{[COMP]}$$

The restriction operator η in $(\eta m, a)(\hat{Sender} \mid \hat{Receiver})$ prevents the environment from interacting with *Sender* and *Receiver* through the channels m and a. For present purposes, it can be taken to be the same as the name restriction operator of [16] for CCS.[1]

The obligation $Sender \subseteq \hat{Sender}$ requires that every behavior of *Sender* is a possible behavior of $\hat{Sender}$. Note that the interaction between the component processes S and U_s has been abstracted away in the specification $\hat{Sender}$. Thus, the state space of $\hat{Sender}$ is smaller than that of *Sender*. However, the problem with [COMP] is that, in fact, $Sender \not\subseteq \hat{Sender}$, since *Sender* can be sending arbitrary messages if acknowledgements arrive at unexpected times, whereas $\hat{Sender}$ ignores spurious acknowledgements. Also, $Receiver \not\subseteq \hat{Receiver}$ for similar reasons. Since these obligations do not hold, the rule [COMP] cannot be used to prove that *System* does not deadlock.

The abstract process $\hat{Sender}$ is a correct abstraction of *Sender* only in an appropriate environment. Similarly, abstract process $\hat{Receiver}$ is a correct abstraction of *Receiver* only in an appropriate environment. The assume-guarantee proof rule shown below, allows the *Sender* and *Receiver* to be analyzed in composition with their abstract environments:

$$\frac{\begin{array}{l} (\eta m, a)(Sender \mid \hat{Receiver}) \subseteq \hat{Sender} \\ (\eta m, a)(\hat{Sender} \mid Receiver) \subseteq \hat{Receiver} \\ (\eta m, a)(\hat{Sender} \mid \hat{Receiver}) \models \psi \end{array}}{(\eta m, a)(Sender \mid Receiver) \models \psi} \quad \text{[AG]}$$

Note that the obligations of the [AG] rule require the *Sender* to conform with $\hat{Sender}$ only in the environment provided by $\hat{Receiver}$. Similarly *Receiver* is required to conform with $\hat{Receiver}$ only in the environment provided by $\hat{Sender}$. Thus, a model checker can discharge the obligations of the [AG] rule and prove deadlock freedom of *System* without having to explore the entire state space of *System* directly. The soundness of such a proof rule requires certain side conditions expressing progress, and is established using an induction over time. In Section 3 we state such side conditions and prove this rule for CCS.

Now suppose that the *Sender* is part of a vendor on the world wide web and the *Receiver* is part of a customer. A common situation is the customer first goes to the vendor's website and after authentication gets fresh channels (these could

[1] The reason we use the notation η is technical and will be explained later.

$S(m,a) = \mu\alpha.(x?.m!.a?.\alpha + a?.(m!)^*)$	$\tau_{S(m,a)} = \mu\alpha.(x?.m!.a?.\alpha + a?.(m!)^*)$
$U_s = \mu\gamma.(x!.\gamma)$	$\tau_{U_s} = \mu\gamma.(x!.\gamma)$
$Sender(m,a) = \nu x.(S(m,a) \mid U_s)$	$\tau_{Sender(m,a)} = \nu x.(\tau_{S(m,a)} \mid \tau_{U_s})$
	$\hat{\tau}_{Sender(m,a)} = \mu\alpha.(m!.a?.\alpha)$
$R(m,a) = \mu\beta.(m?.(y!.a!.\beta + m?.(a!)^*))$	$\tau_{R(m,a)} = \mu\beta.(m?.(y!.a!.\beta + m?.(a!)^*))$
$U_r = \mu\delta.(y?.\delta)$	$\tau_{U_r} = \mu\delta.(y?.\delta)$
$Receiver(m,a) = \nu y.(R(m,a) \mid U_r)$	$\tau_{Receiver(m,a)} = \nu y.(\tau_{R(m,a)} \mid \tau_{U_r})$
	$\hat{\tau}_{Receiver(m,a)} = \mu\beta.(m?.a!.\beta)$
$Vendor(m,a) = www![m,a].Sender(m,a)$	$\tau_{Vendor(m,a)} = www![(m,a)\tau_{Receiver(m,a)}].$ $(\tau_{Sender(m,a)} \mid \tau_{Receiver(m,a)})$
$Customer = www?[m,a].Receiver(m,a)$	$\tau_{Customer} = www?[(m,a)\tau_{Receiver(m,a)}].$ $\tau_{Receiver(m,a)} \uparrow \{m,a\}$
$System = \nu www.((\nu msg, ack.Vendor(msg,ack)) \mid Customer)$	

Fig. 2. An Sender-Reciver System in π-Calculus and Its Process Types.

be fresh URLs) over which the transaction actually happens. Such an interaction can be modeled using the channel name generation and channel passing capabilities of the π-calculus. Figure 2 shows a model of the above scenario in the π-calculus. If we want to check that the vendor process *Vendor* and customer process *Customer* do not deadlock, we need to be able to handle channel passing in our analysis. A promising approach is to first use a type-system to construct first-order approximations of the processes called process-types, and use model checking on the process-types. In Section 4 we build a type-system inspired by the work of Igarashi and Kobayashi to abstract π-calculus processes using CCS processes as process types. The right side of Figure 2 shows the process types generated by the type-system for each π-calculus process on the left. A model checker is used as a subroutine inside the type-checker. In our type system, it turns out that the model checker is asked to check:

$$(\eta msg, ack)(\tau_{Sender(msg,ack)} \mid \tau_{Receiver(msg,ack)}) \models \psi$$

Here $\tau_{Sender(msg,ack)}$ is the process type for the $Sender(msg, ack)$ process and $\tau_{Receiver(msg,ack)}$ is the process type for the $Receiver(msg, ack)$ process. These process types are identical to the *Sender* and *Receiver* processes from Figure 1. The notation $z![\tau].P$ is used for the type of a process which sends a channel along z and continues as P, where τ is a type describing the interactions that could possibly happen on the sent channel. If the user writes behavioral type specifications $\hat{\tau}_{Sender(msg,ack)}$ and $\hat{\tau}_{Receiver(msg,ack)}$ at the module interfaces for $Sender(msg,ack)$ and $Receiver(msg,ack)$ we can use our assume-guarantee rule to mitigate the state-explosion that happens inside the type-checker.

Figure 3 shows a staged-server system [13] with two stages. This example was inspired from a web-crawler example in [14]. *StageA* receives inputs from

$$
\begin{array}{l|l}
Sem = \mu\beta.(acquire!.release?.\beta) & \tau_{Sem} = \mu\beta.(acquire!.release?.\beta) \\
StageA = A?[x].acquire?.(\nu y)B?[y].y?.x! & \tau_{StageA} \\
StageB = \mu\alpha.(B?[y].y!.release!.\alpha) & \quad = A?[(x)x!].acquire?.(\nu y)B?[(y)y!].(y? \mid y!) \\
 & \tau_{StageB} = \mu\alpha.(B?[(y)y!].release!.\alpha) \\
System = (*StageA) \mid StageB^k \mid Sem^k & \\
 & \tau_{System} = (*\tau_{StageA}) \mid {\tau_{StageB}}^k \mid {\tau_{Sem}}^k
\end{array}
$$

Fig. 3. A Staged-Server in π-Calculus and Its Process Types.

the user and then passes each request to *StageB*. When *StageB* responds to the request, the response is first received by *StageA* and then passed on to the user. The system comprises of an unbounded number of copies of *StageA* and k copies of *StageB*. For the purpose of resource control, k copies of a semaphore process *Sem* are used to control access to the k copies of *StageB*. Name generation is used to model matching the requests with appropriate responses. With every request, *StageA* generates a new channel y, sends y to *StageB*, and waits for a response on channel y. The right hand side of Figure 3 shows the process types generated by our type system. The channel passing from *StageA* to *StageB* has been approximated in the process types. Note that *StageA* sends a new channel y to *StageB*. Upon receiving the channel y, *StageB* does $y!$. The type τ_{StageA} does not send any channels to *StageB*. The effect of *StageB* doing $y!$ is statically transferred inside the description of τ_{StageA} by the type system.

The process *System* satisfies the following property: whenever *StageA* wants to send a message to *StageB* after successfully acquiring the semaphore (by executing *acquire*?), then the send $B![y]$ never blocks. Even though τ_{System} is an infinite state system, due to unbounded number copies of τ_{StageA}, we can check this property on τ_{System} by using a model checker with counting abstraction similar to [12].

3 Assume-Guarantee Rule for CCS

In this section we give syntax and semantics of CCS processes, and we define trace containment for such processes. The main result of the section is Theorem 1, which gives an assume-guarantee rule for CCS.

3.1 Syntax and Semantics of CCS Processes

The syntax of CCS processes P is given by the following definition.

$$P ::= \alpha \mid \mathbf{0} \mid \Sigma_i G_i \mid P \mid Q \mid \mu\alpha.P \mid (\nu x)P$$

$$G ::= x!^t.P \mid x?^t.P$$

The structural preorder $\preceq$ is the least reflexive and transitive relation closed under the following rules, together with renaming and reordering of bound variables and reordering of terms in a summation. The notation $P \equiv Q$ abbreviates $P \preceq Q$ and $Q \preceq P$. The set of free names of P is denoted $\mathsf{fn}(P)$.

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

$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad \mu\alpha.P \equiv P[\mu\alpha.P/\alpha]$$

$$\frac{x \notin \mathsf{fn}(P)}{P \mid (\nu x)Q \preceq (\nu x)(P \mid Q)} \qquad \frac{P \preceq P' \quad Q \preceq Q'}{P \mid Q \preceq P' \mid Q'}$$

Figure 4: Structural Preorder

$$P \xrightarrow{\epsilon} P \qquad [\text{EPS}]$$

$$(\ldots + x!^{t_1}.P + \ldots) \mid (\ldots + x?^{t_2}.Q + \ldots) \xrightarrow{x^{t_1,t_2}} P \mid Q \qquad [\text{REACT}]$$

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

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

$$\frac{P \xrightarrow{x^{t_1,t_2}} P'}{(\nu x)P \xrightarrow{\tau^{t_1,t_2}} (\nu x)P'} \quad [\text{TAU}] \qquad \frac{P \xrightarrow{\ell} P' \quad x \notin \ell}{(\nu x)P \xrightarrow{\ell} (\nu x)P'} \quad [\text{RES}]$$

$$\frac{P \preceq P' \quad P' \xrightarrow{\ell} Q' \quad Q' \preceq Q}{P \xrightarrow{\ell} Q} \quad [\text{S-CONG}] \qquad \frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \quad [\text{PAR}]$$

Eta rules

$$\frac{P \xrightarrow{x^{t_1,t_2}} P'}{(\eta x)P \xrightarrow{x^{t_1,t_2}} (\eta x)P'} \quad [\text{ETA1}] \qquad \frac{P \xrightarrow{\ell} P' \quad x \notin \ell}{(\eta x)P \xrightarrow{\ell} (\eta x)P'} \quad [\text{ETA2}]$$

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

Figure 5: Labeled Reduction on CCS Processes

We write $*P$ as an abbreviation for $\mu\alpha.(P \mid \alpha)$, P^* as an abbreviation for $\mu\alpha.(P.\alpha)$, and P^k as an abbreviation for k copies of process P in parallel. Throughout this section, P,Q, P' and Q' range over CCS processes.

We augment the usual syntax of CCS with tags on send and receive operations. Actions in CCS are of the form $x!^t$,$x?^t$, $x!^{t_1,t_2}$, $x?^{t_1,t_2}$, τ^{t_1,t_2} or ϵ. The actions $x!^t$ and $x?^t$ denote commitments and actions $x!^{t_1,t_2}$, $x?^{t_1,t_2}$ denote reactions. Note that commitments have a single tag and reactions have two tags. The action τ^{t_1,t_2} denotes the invisible or silent reaction, and action ϵ denotes the null action.

Figure 5 defines the labeled reduction relation on processes. As indicated by rule S-CONG in Figure 5, reduction is modulo structural congruence, defined in Figure 4. Note that in addition to the usual rules for the restriction operator ν we have rules ETA1 and ETA2 for the restriction operator η. This operator is the same as ν, only with different observability properties: The expression $(\eta\overline{x})P$ is simply meta-notation for a ν-abstraction whose interactions can be observed. This notation is needed to state our assume-guarantee rule.

Sometimes it is insignificant if an action is a send or receive. In such cases, we drop the ? and ! symbol from the action for brevity. Let Act be the set of all actions of the form x^{t_1,t_2}, x^t, τ^{t_1,t_2} or ϵ. We use ω, $\omega_1, \omega_2, \ldots$ to range over finite sequences of actions, and we write $\omega_{[i]}$ to denote the i'th element of ω. If P is a CCS process with

$$P \xrightarrow{\omega_{[0]}} P_1 \xrightarrow{\omega_{[1]}} P_2 \xrightarrow{\omega_{[2]}} \ldots P_{n-1} \xrightarrow{\omega_{[n-1]}} P_n$$

then $\omega = \omega_{[0]}\omega_{[1]}\omega_{[2]}\ldots\omega_{[n-1]}$ is a *trace* of P. In such cases, we lift reductions to sequences of actions and write $P \xrightarrow{\omega} P_n$. The set of all traces of P is denoted $\mathsf{Tr}(P)$. We let $\mathsf{Act}(\omega)$ denote the set of actions occurring in the trace ω, and we define for a process P the set of actions $\mathsf{Act}(P) = \bigcup_{\omega\in\mathsf{Tr}(P)} \mathsf{Act}(\omega)$.

We will assume that, for any set of processes under consideration, tags are used only once, *i.e.*, no tag occurs twice in the processes. We let $\mathsf{T}(P)$ denote the set of tags occurring in P.

Let $\omega = \omega_{[0]}\omega_{[1]}\ldots\omega_{[n]}$ be a trace in $\mathsf{Tr}((\eta\overline{x})(P \mid Q))$, $\omega_{[i]} \in \mathsf{Act}$. For an element $\omega_{[i]}$ in ω we will now define the *projection of* $\omega_{[i]}$ *onto* P, denoted $(\omega_{[i]})_P$, as follows. The definition is by cases over the form of $\omega_{[i]} \in \mathsf{Act}$:

$$(x^{t_1,t_2})_P = \begin{cases} x^{t_1,t_2} & \text{if } t_1 \in \mathsf{T}(P) \text{ and } t_2 \in \mathsf{T}(P) \\ x^{t_1} & \text{if } t_1 \in \mathsf{T}(P) \text{ and } t_2 \notin \mathsf{T}(P) \\ x^{t_2} & \text{if } t_1 \notin \mathsf{T}(P) \text{ and } t_2 \in \mathsf{T}(P) \\ \epsilon & \text{if } t_1 \notin \mathsf{T}(P) \text{ and } t_2 \notin \mathsf{T}(P) \end{cases}$$

$$(\tau^{t_1,t_2})_P = \begin{cases} \tau^{t_1,t_2} & \text{if } t \in \mathsf{T}(P) \text{ and } t_2 \in \mathsf{T}(P) \\ \epsilon & \text{if } t_1 \notin \mathsf{T}(P) \text{ or } t_2 \notin \mathsf{T}(P) \end{cases}$$

$$(x^t)_P = \begin{cases} x^t & \text{if } t \in \mathsf{T}(P) \\ \epsilon & \text{if } t \notin \mathsf{T}(P) \end{cases}$$

$$(\epsilon)_P = \epsilon$$

The projection $(\omega_{[i]})_Q$ is defined analogously. If $\omega = \omega_{[0]}\omega_{[1]}\omega_{[2]}\ldots\omega_{[n-1]}$ is a trace of $P \mid Q$, then we define the *projection of* ω *onto* P, denoted ω_P, to be

given by

$$(\omega_P)_{[i]} = (\omega_{[i]})_P \text{ for } i = 0 \ldots n-1$$

and the projection of ω onto Q, denoted ω_Q, is defined analogously.

We need an operation to combine traces. Let $\oplus$ be the partial function on $\mathsf{Act}\times\mathsf{Act}$, given by $x^t \oplus x^{t'} = x^{t,t'}$ and $\ell \oplus \epsilon = \epsilon \oplus \ell = \ell$ for all $\ell \in \mathsf{Act}$. For traces ω_1 and ω_2 of equal length we define $\omega_1 \oplus \omega_2$ by setting $(\omega_1 \oplus \omega_2)_{[i]} = (\omega_1)_{[i]} \oplus (\omega_2)_{[i]}$. We consider traces *modulo* ϵ, that is, any number of occurrences of ϵ can be tacitly inserted or removed from a trace (hence any trace has some representative of any given length greater than some smallest length). Finally, if $\overline{x}$ is a list of channel names, we define the relation

$$\overline{x} \vdash \ell \sim \ell'$$

to hold if and only if $\ell \oplus \ell'$ is well defined and both of the following conditions are satisfied for all $x \in \overline{x}$:

- if ℓ is of the form x^t, then $\ell' \neq \epsilon$
- if ℓ' is of the form x^t, then $\ell \neq \epsilon$

We lift the relation to traces of equal length n, by defining $\overline{x} \vdash \omega_1 \sim \omega_2$ to hold if and only if for all $i = 0 \ldots n-1$ we have $\overline{x} \vdash (\omega_1)_{[i]} \sim (\omega_2)_{[i]}$.

Lemma 1. *Assume that* $\mathsf{Act}(\omega) \subseteq \mathsf{Act}(A) \cup \mathsf{Act}(B)$. *Then we have:* $\omega \in \mathsf{Tr}(A \mid B)$ *if and only if* $\omega_A \oplus \omega_B$ *is well defined and* $\omega_A \in \mathsf{Tr}(A)$ *and* $\omega_B \in \mathsf{Tr}(B)$ *and* $\omega = \omega_A \oplus \omega_B$.

Lemma 2. *Assume that* $\mathsf{Act}(\omega) \subseteq \mathsf{Act}(A) \cup \mathsf{Act}(B)$. *Then we have:* $\omega \in \mathsf{Tr}((\eta\overline{x})(A \mid B))$ *if and only if* $\overline{x} \vdash \omega_A \sim \omega_B$ *and* $\omega_A \in \mathsf{Tr}(A)$ *and* $\omega_B \in \mathsf{Tr}(B)$ *and* $\omega = \omega_A \oplus \omega_B$.

Note that Lemma 2 coincides with Lemma 1 if $\overline{x}$ is empty.

3.2 Trace Containment

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} or x^t with x. For a trace ω, we define the *norm* of ω, denoted $\mathsf{N}(\omega)$, to be the sequence $(\omega^\tau)^\circ$. We write $\omega =_{\mathsf{N}} \omega'$ as an abbreviation for $\mathsf{N}(\omega) = \mathsf{N}(\omega')$.

We say that a process I is *trace contained in* process S with respect to process P, written $I \subseteq_P S$, if and only if $I \equiv (\eta\overline{x})(P \mid Q)$ and for every $\omega \in \mathsf{Tr}(I)$ there exists $\omega' \in \mathsf{Tr}(S)$ such that $\omega' =_{\mathsf{N}} \omega_P$. We abbreviate $I \subseteq_I S$ as $I \subseteq S$.

Let x be a channel name in $\overline{x}$. We say that x is a *non-blocking* channel of process P in the process $(\eta\overline{x})(P \mid Q)$ if and only if whenever the following conditions hold:

1. $P \xrightarrow{\omega_1} P'$
2. $Q \xrightarrow{\omega_2} Q'$

3. $\overline{x} \vdash \omega_1 \sim \omega_2$
4. $P' \equiv (\ldots + a^{t_1}.P'' + \ldots)$

then we have

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

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

3.3 Assume-Guarantee Rule

Given an implementation I and specification S, suppose we want to check if $I \subseteq S$. Suppose further that $I = (\nu\overline{x})(P \mid Q)$ is a composition of two processes P and Q that interact over a set of channels $\overline{x}$, and that the specification $S = (\nu\overline{x})(P' \mid Q')$ is structurally similar to the implementation I. Then Theorem 1 gives a way of checking if $I \subseteq S$ without exploring the entire state space of I directly.

Theorem 1. *(Assume-Guarantee) For any processes P, Q, P', Q' suppose*

A1. $(\eta\overline{x})(P \mid Q') \subseteq_P P'$
A2. $(\eta\overline{x})(P' \mid Q) \subseteq_Q Q'$
A3. *Every channel x in $\overline{x}$ is either non-blocking for P in $(\eta\overline{x})(P \mid Q')$ or non-blocking for Q in $(\eta\overline{x})(P' \mid Q)$.*

Then we have

$$(\eta\overline{x})(P \mid Q) \subseteq (\eta\overline{x})(P' \mid Q')$$

Before proving the theorem we state a few lemmas. In the following, we will sometimes use process superscripts on traces, as in ω^P. This is a naming convention intended as a help to remind the reader that ω^P is in $\mathsf{Tr}(P)$.

Lemma 3. *If $\overline{x} \vdash \omega_1 \sim \omega_2$ and $\omega_1 =_N \omega_1'$, then $\overline{x} \vdash \omega_1' \sim \omega_2$.*

Lemma 4. *If $\omega_1 \oplus \omega_2$ is well defined and $\omega_1 =_N \omega_1'$, then $\omega_1' \oplus \omega_2$ is well defined and $\omega_1 \oplus \omega_2 =_N \omega_1' \oplus \omega_2$.*

Lemma 5. *Suppose that*

1. $\omega \in \mathsf{Tr}((\eta\overline{x})(P \mid Q))$
2. $\omega^{P'} \in \mathsf{Tr}(P')$ *with* $\omega^{P'} =_N \omega_P$
3. $(\eta\overline{x})(P' \mid Q) \subseteq_Q Q'$

Then there exists $\omega^{Q'} \in \mathsf{Tr}(Q')$ such that $\omega^{Q'} =_N \omega_Q$.

For natural numbers k, we can talk about *trace containment up to k*, denoted $\subseteq_P^k$, by defining $(\eta\overline{x})(P \mid Q) \subseteq_P^k P'$ if and only if for all traces $\omega \in \mathsf{Tr}((\eta\overline{x})(P \mid Q))$ of length at most k, there exists $\omega' \in \mathsf{Tr}(P')$ with $\omega' =_N \omega_P$.

Lemma 6. *Suppose, for any k, that*

1. $(\eta\overline{x})(P \mid Q) \subseteq_P^k P'$
2. $(\eta\overline{x})(P \mid Q) \subseteq_Q^k Q'$

Then $(\eta\overline{x})(P \mid Q) \subseteq^k (\eta\overline{x})(P' \mid Q')$.

Lemma 7. *Let* $\omega \in \mathsf{Tr}((\eta\overline{x})(P \mid Q))$ *and let* $\omega^{P|Q'} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q'))$ *such that*

1. $(\eta\overline{x})(P \mid Q') \subseteq_P P'$
2. $\omega_P =_{\mathsf{N}} (\omega^{P|Q'})_P$
3. $\omega^{P|Q'}.\tilde{\omega} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q'))$

Then there exists $\omega^{P'} \in \mathsf{Tr}(P')$ *such that* $\omega^{P'} =_{\mathsf{N}} \omega_P.(\tilde{\omega})_P$.

Lemma 8. *(Context Substitution)*
Assume

$$(\eta\overline{x})(P \mid Q) \xrightarrow{\omega} (\eta\overline{x})(P_k \mid Q_k)$$

and $\omega^{P|Q'} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q'))$ *with* $\omega =_{\mathsf{N}} \omega^{P|Q'}$ *and* $\omega_P = (\omega^{P|Q'})_P$. *Then*

$$(\eta\overline{x})(P \mid Q') \xrightarrow{\omega^{P|Q'}} (\eta\overline{x})(P_k \mid Q'_k)$$

for some Q'_k.

We are now ready to prove Theorem 1.

Proof. Assuming A1, A2 and A3 we prove by induction on the length of traces in $\mathsf{Tr}((\eta\overline{x})(P \mid Q))$ the stronger conclusion

B1. $(\eta\overline{x})(P \mid Q) \subseteq (\eta\overline{x})(P' \mid Q')$ and
B2. $(\eta\overline{x})(P \mid Q) \subseteq_P P'$ and
B3. $(\eta\overline{x})(P \mid Q) \subseteq_Q Q'$.

Let $\omega = \omega_0\omega_1 \ldots \omega_{k-1}\omega_k \ldots \omega_n$ be an arbitrary trace in $\mathsf{Tr}((\eta\overline{x})(P \mid Q))$. Let $\omega_{\leq i}$ denote the prefix $\omega_0\omega_1 \ldots \omega_i$, for $0 \leq i \leq n$. We assume the induction hypothesis holds for $\omega_{\leq k}$ and prove it for $\omega_{\leq(k+1)}$.

We first establish the following:

C1. $\omega_{\leq i} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q))$ for all $0 \leq i \leq n$.
C2. $\exists \omega_k^{P'|Q} \in \mathsf{Tr}((\eta\overline{x})(P' \mid Q))$. $\omega_{\leq k} =_{\mathsf{N}} \omega_k^{P'|Q} \wedge (\omega_{\leq k})_Q = (\omega_k^{P'|Q})_Q$.
C3. $\exists \omega_k^{P|Q'} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q'))$. $\omega_{\leq k} =_{\mathsf{N}} \omega_k^{P|Q'} \wedge (\omega_{\leq k})_P = (\omega_k^{P|Q'})_P$.

C1 follows by the assumptions and the definition of $\omega_{\leq i}$. To prove C2, we first observe that, because $\omega_{\leq k} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q))$, Lemma 2 shows that $(\omega_{\leq k})_P \in \mathsf{Tr}(P)$ and $(\omega_{\leq k})_Q \in \mathsf{Tr}(Q)$ and

$$\overline{x} \vdash (\omega_{\leq k})_P \sim (\omega_{\leq k})_Q \tag{1}$$

By induction hypothesis (B2) applied to $\omega_{\leq k}$ we get

$$\exists \omega_k^{P'} \in \mathsf{Tr}(P').\ (\omega_{\leq k})_P =_{\mathsf{N}} \omega_k^{P'} \tag{2}$$

Now, choose $\omega_k^{P'}$ according to (2). By (1), (2) and Lemma 3 we have

$$\overline{x} \vdash \omega_k^{P'} \sim (\omega_{\leq k})_Q \tag{3}$$

Define $\omega_k^{P'|Q}$ by setting

$$\omega_k^{P'|Q} = \omega_k^{P'} \oplus (\omega_{\leq k})_Q$$

Then $\omega_k^{P'|Q}$ is well defined and in $\mathsf{Tr}((\eta\overline{x})(P' \mid Q))$ by (3) and Lemma 2. Because (by (2)) we have $\omega_k^{P'} =_{\mathsf{N}} (\omega_{\leq k})_P$, it follows from Lemma 4 that

$$\omega_k^{P'} \oplus (\omega_{\leq k})_Q =_{\mathsf{N}} (\omega_{\leq k})_P \oplus (\omega_{\leq k})_Q$$

which shows that

$$\omega_k^{P'|Q} =_{\mathsf{N}} \omega_{\leq k} \tag{4}$$

Furthermore, it follows from the definition of $\omega_k^{P'|Q}$ that

$$(\omega_{\leq k})_Q = (\omega_k^{P'|Q})_Q \tag{5}$$

It follows from (4) and (5) that $\omega_k^{P'|Q}$ is a witness of C2. This concludes the proof of C2. The claim C3 is proven by a symmetric argument, using induction hypothesis (B3).

We now proceed to prove B1, B2 and B3 for the inductive case $k+1$ by a case analysis on the form of ω_{k+1}. For space reasons we prove only one representative case. The full proof can be found in the technical report [20].

– **Case 1**. ω_{k+1} is an interaction x^{t_1,t_2}, for $x \in \overline{x}$: WLOG let $t_1 \in \mathsf{T}(P)$, $t_2 \in \mathsf{T}(Q)$, and let x be non-blocking for P in $(\eta\overline{x})(P \mid Q')$.
We know from C3 that, for some $\omega_k^{P|Q'} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q'))$ we have

$$\omega_{\leq k} =_{\mathsf{N}} \omega_k^{P|Q'} \text{ and } (\omega_{\leq k})_P = (\omega_k^{P|Q'})_P \tag{6}$$

By our assumptions, $\omega_{k+1} = x^{t_1,t_2}$, so that P can make the commitment x^{t_1} in step ω_{k+1}. Hence, we have

$$(\eta\overline{x})(P \mid Q) \xrightarrow{\omega_{\leq k}} (\eta\overline{x})(P_k \mid Q_k) \xrightarrow{x^{t_1,t_2}} (\eta\overline{x})(P_{k+1} \mid Q_{k+1})$$

with $P_k \xrightarrow{x^{t_1}} P_{k+1}$, for some $P_k, Q_k, P_{k+1}, Q_{k+1}$. By (6) together with Lemma 8 we can conclude that

$$(\eta\overline{x})(P \mid Q') \xrightarrow{\omega_k^{P|Q'}} (\eta\overline{x})(P_k \mid Q'_k)$$

for some Q'_k. Because x is non-blocking for P in $(\eta\overline{x})(P \mid Q')$, it follows that

$$Q'_k \xrightarrow{\tau^*} Q'' \xrightarrow{x^{t_3}} Q'_{k+1}$$

for some sequence of τ-actions τ^* and some t_3, Q'', Q'_{k+1}. Putting the previous results together, we can conclude that

$$\begin{array}{l}(\eta\overline{x})(P \mid Q') \xrightarrow{\omega_k^{P|Q'}} \\ (\eta\overline{x})(P_k \mid Q'_k) \xrightarrow{\tau^*} \\ (\eta\overline{x})(P_k \mid Q'') \xrightarrow{x^{t_1,t_3}} \\ (\eta\overline{x})(P_{k+1} \mid Q_{k+1})\end{array}$$

Hence, we have

$$\omega_k^{P|Q'}.\tau^*.x^{t_1,t_3} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q')) \tag{7}$$

By (6), (7) and assumption A1, Lemma 7 is applicable (taking $\tilde{\omega} = \tau^*.x^{t_1,t_2}$), and we get that there exists $\omega^{P'} \in \mathsf{Tr}(P')$ such that

$$\omega^{P'} =_{\mathsf{N}} (\omega_{\leq k})_P.(\tau^*.x^{t_1,t_3})_P =_{\mathsf{N}} (\omega_{\leq k})_P.x^{t_1} = (\omega_{\leq k+1})_P$$

Hence, we have

$$\omega^{P'} =_{\mathsf{N}} (\omega_{\leq k+1})_P \tag{8}$$

thereby showing B2 for the inductive step $k+1$, as witnessed by $\omega^{P'}$ in (8). Since $\omega_{\leq k+1} \in \mathsf{Tr}((\eta\overline{x})(P \mid Q))$, it follows from (8) together with A2 via Lemma 5 that there exists $\omega^{Q'} \in \mathsf{Tr}(Q')$ with $\omega^{Q'} =_{\mathsf{N}} (\omega_{\leq k+1})_Q$. This shows B3 for the inductive step $k+1$. Lemma 6 applied to B2 and B3 for $k+1$ then yields B1 for the step $k+1$.

– **Remaining cases**. See technical report [20].

□

A model checker can discharge obligations A1, A2 and A3 and reach the desired conclusion. Note that

$$(\eta\overline{x})(P \mid Q) \subseteq (\eta\overline{x})(P' \mid Q') \quad \Rightarrow \quad (\nu\overline{x})(P \mid Q) \subseteq (\nu\overline{x})(P' \mid Q')$$

Thus, η is just a meta-process notation that lets us state the obligations A1, A2 and A3, all of which require observing the interactions on channels in $\overline{x}$. We note that model checking CCS is undecidable in general. However model checking is decidable for certain fragments of CCS such as the finite control

In the reduction rules below, the structural preorder $\preceq$ is as defined in Figure 4 with the additional rule $*P \preceq *P \mid P$.

Syntax

$$P ::= \mathbf{0} \quad | \quad \Sigma_i G_i \quad | \quad P \mid Q \quad | \quad *P \quad | \quad (\nu x)P$$

$$G ::= x!^t[\overline{y}].P \quad | \quad x?^t[\overline{y}].P$$

Semantics

$$(\ldots + x!^t[\overline{z}].P + \ldots) \mid (\ldots + x?^{t'}[\overline{y}].Q + \ldots) \xrightarrow{x^{t,t'}} P \mid [\overline{z}/\overline{y}]Q \qquad \text{[R-COM]}$$

$$\frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \quad \text{[R-PAR]} \qquad \frac{P \preceq P' \quad P' \xrightarrow{\ell} Q' \quad Q' \preceq Q}{P \xrightarrow{\ell} Q} \quad \text{[R-SP-CONG]}$$

$$\frac{P \xrightarrow{x^{t_1,t_2}} P'}{(\nu x)P \xrightarrow{\tau^{t_1,t_2}} (\nu x)P'} \quad \text{[R-NEW1]} \qquad \frac{P \xrightarrow{\ell} P' \quad x \notin \ell}{(\nu x)P \xrightarrow{\ell} (\nu x)P'} \quad \text{[R-NEW2]}$$

Figure 6: Syntax and Semantics of π Calculus

fragment, which disallows μ recursion inside parallel composition, and the ν free fragment. In order to model check an arbitrary CCS process one could first construct an abstraction that falls in such a decidable fragment and then model check the abstraction.

Theorem 1 generalizes to the case of any finite parallel composition $\prod_i P_i$. Since the proof method is the same as shown in the proof of Theorem 1, we confine ourselves to recording

Theorem 2. *(Assume-Guarantee) Let $\xi_i = P'_1 \mid \ldots \mid P'_{i-1} \mid P'_{i+1} \mid \ldots \mid P'_n$, for $i = 1 \ldots n$. Then the following inference rule is sound*

$$\frac{\forall i.\ (\eta\overline{x})(P_i \mid \xi_i) \subseteq_{P_i} P'_i}{(\eta\overline{x})\textstyle\prod_i P_i \subseteq (\eta\overline{x})\prod_i P'_i}(*)$$

provided that the side-condition $(*)$ is satisfied:

$$(*) \begin{cases} \forall x \in \overline{x}.\ \forall i, j. \\ \textit{either } x \textit{ non-blocking for } P_i \textit{ in } (\eta\overline{x})(P_i \mid \xi_i) \\ \textit{or } x \textit{ non-blocking for } P_j \textit{ in } (\eta\overline{x})(P_j \mid \xi_j) \end{cases}$$

4 A Behavioral Module System for π-Calculus

The syntax and semantics of the π calculus is shown in Figure 6. We use abstract processes Γ for types and type environments. A type judgment in this system is of the form $\Gamma \rhd P$, meaning that the abstract process Γ is a correct abstraction of the concrete π calculus process P. In the sequel, we will refer to abstract

$$\mathbf{0} \rhd \mathbf{0} \quad \text{[T-Zero]} \qquad \frac{\gamma_i \rhd G_i \text{ for } i = 1 \ldots n}{\gamma_1 + \ldots + \gamma_n \rhd G_1 + \ldots + G_n} \quad \text{[T-Choice]}$$

$$\frac{\Gamma_1 \rhd P_1 \quad \Gamma_2 \rhd P_2}{\Gamma_1 \mid \Gamma_2 \rhd P_1 \mid P_2} \quad \text{[T-Par]} \qquad \frac{\Gamma_1 \rhd P}{x!^t[(\overline{y})\Gamma_2].(\Gamma_1 \mid [\overline{z}/\overline{y}]\Gamma_2) \rhd x!^t[\overline{z}].P} \quad \text{[T-Out]}$$

$$\frac{\Gamma \rhd P}{*\Gamma \rhd *P} \quad \text{[T-Rep]} \qquad \frac{\Gamma \rhd P}{x?^t[(\overline{y})(\Gamma \uparrow_{\mathcal{V}-\overline{y}})].(\Gamma \uparrow_{\overline{y}}) \rhd x?^t[\overline{y}].P} \quad \text{[T-In]}$$

$$\frac{\Gamma \rhd P \quad \Gamma \leq \Gamma'}{\Gamma' \rhd P} \quad \text{[T-Sub]} \qquad \frac{\Gamma \rhd P \quad WF(\Gamma \uparrow_{\mathcal{V}-\overline{x}}) \quad (\eta\overline{x})\Gamma \uparrow_{\mathcal{V}-\overline{x}} \models \Box\psi}{(\nu\overline{x})\Gamma \rhd (\nu\overline{x})P} \quad \text{[T-New]}$$

Figure 7: Typing Rules

processes Γ as *process types*. Throughout the remainder of this section, P, P_i and P' range over π-calculus processes.

Our type system is a variant of the system presented by Igarashi and Kobayashi [11], with the primary difference being that our process types are exactly CCS. The process types of Igarashi and Kobayashi form a subcalculus of CCS, because they do not include the name restriction operator, ν. The inclusion of name restriction enables us to type processes more precisely. Consider the following example process $P = (\nu cd)P'$ where P' is the process

$$\begin{pmatrix} c?.d!+ \\ c!.d! \end{pmatrix} \mid (\nu x) \begin{pmatrix} x!^{t_1} \mid x?^{t_2}.c!^{t_4} \mid \\ x?^{t_3}.c?^{t_5} \end{pmatrix} \mid \left(d?^{t_6}\right)$$

The type of P' in the Igarashi-Kobayashi type system is the process type Γ given by

$$\begin{pmatrix} c?.d!+ \\ c!.d! \end{pmatrix} \mid \begin{pmatrix} t_1 \mid t_2.c!^{t_4} \mid \\ t_3.c?^{t_5} \end{pmatrix} \mid \left(d?^{t_6}\right)$$

In this type, the restriction $(\nu x)\ldots$ has been elided, and all occurrences of x are replaced by tags t_1, t_2, t_3, with $t.\Gamma'$ reducing to Γ'. This is an abstraction of name restriction, and it introduces an overapproximation of the concrete semantics. In this case, the type Γ contains an execution where the receive $d?^{t_6}$ blocks for ever, which arises when reductions on t_2 and t_3 are followed by a reaction between $c!^{t_4}$ and $c?^{t_5}$. However, in process P all executions result in a successful interaction on channel d. In contrast, since our process types contain name restriction, the process type of P is identical to P in our system.

The process types in our type system are defined by the following syntax:

$$\begin{array}{rll} \tau \text{ (tuple types)} & ::= & (x_1, x_2, \ldots, x_n)\Gamma \\ \Gamma \text{ (process types)} & ::= & \mathbf{0} \mid \alpha \mid \gamma_1 + \ldots + \gamma_n \mid (\Gamma_1 \mid \Gamma_2) \mid \mu\alpha.\Gamma \mid (\nu x)\Gamma \\ \gamma & ::= & x!^{\mathbf{t}}[\tau].\Gamma \mid x?^{\mathbf{t}}[\tau].\Gamma \end{array}$$

This language is equivalent to CCS, with typed channels. Channel types do not influence reduction semantics of process types. However, they are used by the

type system to model higher-order message passing in π calculus. The reduction semantics of process types is given by Figure 5. For given process type Γ, each tag t uniquely determines an occurrence of either $x!^t[\tau]$ or $x?^t[\tau]$. In the context of Γ let $\mathcal{T}(t)$ denote the type τ thus associated with t. We abbreviate $x!^t[\mathbf{0}]$ as $x!^t$.

The typing rules of our type system are shown in Figure 7. The type system includes subtyping in rule [T-Sub]. Our subtyping relation $\leq$ is *weak simulation* [16], defined as $\Gamma_1 \leq \Gamma_2$ if and only if for all action sequences ω, whenever $\Gamma_1 \xrightarrow{\omega} \Gamma_1'$, then there exists ω' and Γ_2' such that $\Gamma_2 \xrightarrow{\omega'} \Gamma_2'$ with $\omega =_{\mathsf{N}} \omega'$ and $\Gamma_1' \leq \Gamma_2'$. This definition of $\leq$ satisfies the axioms for a *proper subtyping relation* as defined by Igarashi and Kobayashi [11]. The rule for name restriction [T-New] and the rule for input [T-In] use the *anonymization operator* $\Gamma \uparrow_S$, where S is a set of channel names. The formula $\Box\psi$ in rule [T-New] refers to an invariant, such as deadlock freedom. In order to define the anonymization operator, we first define the *type elimination operator* $\Gamma\backslash_S$, where S is a set of channel names, and it is defined by

$$
\begin{aligned}
\mathbf{0}\backslash_S &= \mathbf{0} \\
\alpha\backslash_S &= \alpha \\
(x?^{\mathbf{t}}[\tau].\Gamma)\backslash_S &= \begin{cases} x?^{\mathbf{t}}[\tau].(\Gamma\backslash_S) & \text{if } x \notin S \\ x?^{\mathbf{t}}.(\Gamma\backslash_S) & \text{otherwise} \end{cases} \\
(x!^{\mathbf{t}}[\tau].\Gamma)\backslash_S &= \begin{cases} (x!^{\mathbf{t}}[\tau].\Gamma)\backslash_S & \text{if } x \notin S \\ x!^{\mathbf{t}}.(\Gamma\backslash_S) & \text{otherwise} \end{cases} \\
(\gamma_1 + \cdots \gamma_n)\backslash_S &= (\gamma_1\backslash_S) + \cdots + (\gamma_n\backslash_S) \\
(\gamma_1 \mid \gamma_n)\backslash_S &= (\gamma_1\backslash_S) \mid (\gamma_n\backslash_S) \\
(\gamma_1 \& \gamma_n)\backslash_S &= (\gamma_1\backslash_S) \& (\gamma_n\backslash_S) \\
(\mu\alpha.\Gamma)\backslash_S &= \mu\alpha.(\Gamma\backslash_S) \\
((\nu\overline{x})\Gamma)\backslash_S &= (\nu\overline{x})(\Gamma\backslash_{S-\overline{x}})
\end{aligned}
$$

Note that the type elimination operator leaves ν-bound names intact. For any set of channels S, let $\mathcal{G}(S)$ be the most general environment on channels S defined by:

$$\mathcal{G}(S) = \mu\alpha.(\sum_{x \in S}(x! + x?)).\alpha$$

The *anonymization operator* $\Gamma \uparrow_S$ is defined as: [2]

$$\Gamma \uparrow_S = (\nu S).(\Gamma\backslash_S \mid \mathcal{G}(S))$$

The rule [T-New] uses the predicate $WF(\Gamma)$ which is defined to hold if and only if for all traces $\omega = \omega_0\omega_1 \ldots \omega_n \in \mathsf{Tr}(\nu\mathcal{V}.\Gamma)$, for all $0 \leq i \leq n$, if $\omega_i = \tau^{t_1,t_2}$ then $\mathcal{T}(t_1) \geq \mathcal{T}(t_2)$.

The rule [T-New] is parameterized on the formula $\Box\psi$. The formula expresses a safety property on the channels $\overline{x}$. For example, it could express the

[2] This definition is equivalent to the definition provided by Igarashi and Kobayashi. Our definition allows refining $\mathcal{G}(S)$ using flow computation, in order to improve precision.

invariant about deadlock freedom on channels in $\overline{x}$. Discharging this assumption will require CCS-model checking of the process type $(\eta\overline{x})\Gamma \upharpoonright_{\mathcal{V}-\overline{x}}$.

We now state a subject reduction theorem, similar to the one in [11]. A proof of the theorem can be found in the technical report [20].

Theorem 3 (Subject Reduction). *If $\Gamma \rhd P$ and $P \xrightarrow{\ell} P'$ with $WF(\Gamma)$, then there exists Γ' such that $\Gamma \xrightarrow{\ell} \Gamma'$ and $\Gamma' \rhd P'$.*

Type checking in our type system requires a model checking step to discharge the assumption

$$(\eta\overline{x})\Gamma \upharpoonright_{\mathcal{V}-\overline{x}} \models \Box\psi \tag{9}$$

in the rule [T-New]. Since our types are CCS processes, Theorem 1 applies. We can therefore alleviate the state space explosion in the model checker using user specified types. More precisely, Theorem 1 shows that the following inference rule is sound for discharging the assumption (9) when Γ is a composition $\Gamma = \Gamma_1 \mid \Gamma_2$:

$$\frac{\begin{array}{l}(\eta\overline{x})(\Gamma_1' \mid \Gamma_2') \models \Box\psi \\ (\eta\overline{x})(\Gamma_1 \mid \Gamma_2') \subseteq_{\Gamma_1} \Gamma_1' \\ (\eta\overline{x})(\Gamma_1' \mid \Gamma_2) \subseteq_{\Gamma_2} \Gamma_2'\end{array}}{(\eta\overline{x})(\Gamma_1 \mid \Gamma_2) \models \Box\psi} \quad \text{[AG]}$$

provided that every channel x in $\overline{x}$ is either non-blocking for Γ_1 in $(\eta\overline{x})(\Gamma_1 \mid \Gamma_2')$ or non-blocking for Γ_2 in $(\eta\overline{x})(\Gamma_1' \mid \Gamma_2)$.

The behavioral types Γ_1' and Γ_2' are user-specified types, analogous to user-specified type signatures in a type system such as ML. In order to apply the [AG] rule, a model checker is needed to discharge the assumptions of the rule and its side-condition. However, the types Γ_1' and Γ_2' are typically more abstract than Γ_1 and Γ_2 and consequently, the state spaces of Γ_1' and Γ_2' could be much smaller than the state spaces of Γ_1 and Γ_2. Thus, using the [AG] rule helps us avoid exploring the state-space of $\Gamma_1 \mid \Gamma_2$, thereby alleviating state explosion. Indeed, if the program subjected to type checking is well modularized, this may save an exponential amount of work.

5 Related Work

Several behavioral type systems have been proposed recently in which types are process-like structures, including [18,22,19,21,11]. Also, other analyses have been proposed to check behavioral properties of concurrent programs, including [8,9,7].

Our work was foremostly inspired by the generic type system of Igarashi and Kobayashi [11]. While Igarashi and Kobayashi use a ν-free fragment of CCS for their process types, our type system uses the entire CCS. In particular, the presence of hiding in the form of name restriction in the process types improves

precision, and opens up several opportunities for modular type checking by exploiting hiding. We use an assume-guarantee principle to discharge the safety check at name restriction (rule [T-New]) in a modular way. Using this technique, we can exploit abstract behavioral specifications. The use of this principle in the context of behavioral type systems appears to be new.

Model checking CCS processes is undecidable in general. However, decidable fragments of CCS have been identified by either disallowing parallel composition under recursion or by disallowing name restriction [4]. Tools have been built to perform bisimulation checking, refinement and model checking of such decidable fragments of CCS [5].

Assume-guarantee rules that allow apparently circular assumptions about operating contexts can be traced back to [17,1,2]. Recent work has used such techniques to model check large hardware circuits [3,15,6,10]. However, all these rules require a nonblocking assumption on the process calculus, and are not directly applicable for model checking CCS. Our assume-guarantee rule for CCS requires progress as a side condition that needs to be checked using the model checker, and, to the best of our knowledge, our soundness result for assume-guarantee reasoning in CCS is new.

6 Conclusion

Checking behavioral properties of concurrent, message-passing programs is an important and difficult problem in todays distributed programming environment. The major obstacle for doing this in practice is the state-explosion problem inherent in model checking. Previous work in model checking strongly suggests that solving this problem requires abstraction and modular methods, so that one can check a large system by checking parts of the system and combine the results.

In sequential languages such as ML, module systems have proven to be very successful for providing both abstraction and modularity. However, the sequential notion of a module system cannot be directly applied to checking behavioral types of concurrent programs, because it is much harder to define what a module boundary means in concurrent contexts. In particular, few interesting properties of concurrent programs are satisfied independent of their intended context of use. Hence, it appears that new principles of modularity are needed for behavioral type systems.

In this paper, we have proposed that assume-guarantee reasoning is a key principle for modular behavioral type checking. Using the assume-guarantee principle, the behavior of a module is precisely guaranteed only under assumptions about its concurrent context. If we want to combine two modules, they can be checked under apparently circular assumptions on each others behavioral "signatures" (circularity is resolved by an induction over time). If programs are well modularized, this principle can lead to an exponential speed up of type checking. Furthermore, this principle suggests ways in which users can provide abstract behavioral specifications at module boundaries.

CCS processes have been proposed as behavioral types for π calculus programs. In order to enable assume-guarantee reasoning for a general class of behavioral type systems, we have proven the assume-guarantee principle sound for CCS with respect to trace containment. To the best of our knowledge, this result is new for CCS. Prior assume-guarantee results require non-blocking semantics on the process calculus and hence cannot be directly applied to CCS. We have shown how this result can be integrated into a particular type system for the π calculus, thereby enabling modular behavioral type checking for message-passing programs. Hiding in the form of name restriction permits writing modular programs in the π calculus. Our type system exploits hiding to decompose the type checking problem.

Much work remains to be done to apply our results to a realistic programming language. In addition to handling the variety of constructs in a realistic language, we need to provide a natural way for the programmer to write behavioral specifications. A realistic system will require a combination of automation (type inference and model checking) and user-annotations (behavioral specifications), and it must allow important programming idioms to type.

Acknowledgements

We thank Greg Meredith for several interesting discussions on behavioral type systems, and on the π-calculus. We also would like to thank Andreas Podelski for comments on this paper.

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. S. Christensen, Y. Hirshfeld, and F. Moller. Decidable subsets of CCS. *The Computer Journal*, 37(4):233–242, 1994.
5. R. J. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.
6. A. 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.
7. J. Feret. Confidentiality analysis of mobile systems. In *SAS'00: Static Analysis Symposium*, LNCS 1824, pages 135–154. Springer-Verlag, 2000.
8. C. Flanagan and M. Abadi. Types for safe locking. In *ESOP'99: European Symposium on Programming*, LNCS 1576, pages 91–108. Springer-Verlag, 1999.
9. 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.
10. 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.

11. 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.
12. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.
13. J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. Technical Report MSR-TR-2001-39, Microsoft Research, 2001.
14. L. McDowell. Tappan: The asynchronous programming language specification and analysis system. Summer-Intern Project Report, Microsoft Research, 2000.
15. K. L. McMillan. A compositional rule for hardware design refinement. In *CAV'97: Computer-Aided Verification*, LNCS 1254, pages 24–35. Springer-Verlag, 1997.
16. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
17. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.
18. 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.
19. 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.
20. S. K. Rajamani and J. Rehof. A behavioral module system for the Pi-calculus. Technical report, Microsoft Research, 2001.
21. A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR'00: Concurrency Theory*, LNCS 1877, pages 474–488. Springer-Verlag, 2000.
22. N. Yoshida. Graph types for monadic mobile processes. In *FSTTCS: Software Technology and Theoretical Computer Science*, LNCS 1180, pages 371–387. Springer-Verlag, 1996.