HAL open science

Typing Communicating Component Assemblages

Michaël Lienhardt, Alan Schmitt, Jean-Bernard Stefani

To cite this version:

Michaël Lienhardt, Alan Schmitt, Jean-Bernard Stefani. Typing Communicating Component Assemblages. 7th International Conference on Generative Programming and Component Engineering (GPCE'08), Oct 2008, Nashville, United States. pp.125–136, 10.1145/1449913.1449933 . inria-00492749

HAL Id: inria-00492749

https://inria.hal.science/inria-00492749

Submitted on 16 Jun 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Typing Communicating Component Assemblages

Michael Lienhardt
Université Grenoble I, France
michael.lienhardt@inria.fr

Alan Schmitt
INRIA, France
alan.schmitt@inria.fr

Jean-Bernard Stefani
INRIA, France
jean-bernard.stefani@inria.fr

Abstract

Building complex component-based software architectures can lead to subtle assemblage errors. In this paper, we introduce a type-system-based approach to avoid message handling errors when assembling component-based communication systems. Such errors are not captured by classical type systems of host programming languages such as Java or ML. Our approach relies on the definition of a small process calculus that captures the operational essence of our target component-based framework for communication systems, and on the definition of a novel type system that combines row types with process types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Components, type system, component types, process types, type inference, assemblage errors, communication systems

1. Introduction

Building software systems from components has many benefits compared to less modular approaches [36]: easier design and development, easier adaptation, maintenance, and evolution. However, constructing a system from components can give rise to non trivial assemblage errors, i.e. errors which occur at run-time because of a faulty assembly of components, that are not captured by classical type systems of the programming languages used for implementation, such as C++, Java or ML. In particular, as noted e.g. in [20], this is the case with communication systems built with dedicated component-based frameworks such as Appia [24], Click [25], Coyote [5], Dream [17], or Ensemble [37]. These frameworks comprise many components (sometimes called *micro-protocols*), that encapsulate low-level system code. Assembling micro-protocols can give rise to subtle errors, in particular errors arising because of incompatible manipulation of protocol data units in different components. These errors are hard to catch because they may be purely the result of a faulty assemblage, and may arise even if individual components are correct.

Dealing with assemblage errors in communication systems and middleware has already been approached in two ways. First, using a theorem prover to formally specify the expected behavior of individual components, and to prove correctness properties on component assemblages, as in Ensemble [20]. Second, using an architecture description language (ADL) to specify component behaviors and assemblage constraints (typically, component dependencies), and to automatically verify the assemblage consistency, as is proposed e.g. in Aster [14], Knit [32], or Plastik [15]. The former approach is comprehensive and can address arbitrary properties, but it requires theorem-proving expertise, which is unlikely to be readily available for systems programmers. The latter approach is more automatic, but it typically supports a limited set of architectural constraints, and a limited set of behavioral checks that fail to address subtler run-time errors such as data manipulation ones.

In this paper we present an approach that extends and complements the ADL-based approach to deal with certain errors that may occur in ill-formed assemblages. Our approach relies on the extension of an ADL with a domain-specific type system, tailored for capturing a class of targeted errors. More specifically, it comprises:

- The definition of a simple process calculus that allows to specify an operational *model* of the target component assemblage (and where program execution is abstracted by a *reduction* relation).
- The definition of a *type system*, that operates on programs abstracted as terms of the process calculus, and which ensures that typable assemblages do not exhibit the targeted class of errors.

In this paper, we illustrate this approach with the handling of data manipulation errors that can occur when building incorrect assemblages using the Dream framework [17]. Dream is interesting because it provides one of the more fine-grained frameworks for building communication systems, with constructs that generalize or subsume those in other communication frameworks such as Appia, Click, or Coyote. However, our approach is not limited to Dream only: the same calculus and type system can be applied e.g. to Appia [24], Click [25] and Coyote [5].

Technically, the paper makes the following contributions. We define a simple process calculus, which constitutes a subset of the Kell calculus [34], to model the behavior and assemblage of Dream components. We define a novel type system which combines *rows* [33], to handle structured messages that are manipulated by Dream components, and *process types* [39, 22] to handle data flows that occur during the execution of component assemblages. We prove that type inference in this type system is undecidable, but we define a semi-algorithm for type inference, that makes our type system usable in practice.

The paper is organized as follows. Section 2 presents a brief overview of the Dream framework, and an informal illustration of the data manipulation errors we target. Section 3 presents the calculus used to model the behavior and assemblage of Dream components. Section 4 describes our type system and its properties. Section 5 presents our semi-inference algorithm. Section 6 discusses related work, and Section 7 concludes the paper with a discussion of further research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.
Copyright © 2008 ACM 978-1-60558-267-2/08/10…$5.00

2. Dream Overview

Dream is a component-based framework, designed for the construction of communication systems (protocol stacks, communication subsystems of middleware for distributed execution). It is constructed using the Java implementation of the Fractal component model [7]. Components in FRACTAL are structured in two parts. The *membrane* part which exposes the input and output *interfaces* (or *ports*) of the component. The *content* part which defines the internal structure of the component. This part can either be actual Java code, or an assemblage of components, whose interfaces are bound via explicit *bindings* (or *connectors*).

The primary data structure in Dream is called a *message*. Messages are used to implement protocol data units (i.e. the data that communication protocols exchange during their execution). Messages are exchanged between Dream components through input and output interfaces which can function in a push mode or a pull mode. A message consists of a list of *labeled chunks* which can be any Java objects, including messages. Within a Dream component, messages can be freely manipulated, and basic operations, like removing, adding or accessing a chunk are provided.

The Dream framework comprises a library of components that encapsulate functions and behaviors commonly found in communication subsystems. These include for instance: *message queues*, which are used to store messages; *transformers*, that have a single input and a single output interface, and that transform a message received on their input interface and deliver the resulting message to their output interface; *routers*, that forward messages received on their single input interface to one or several output interfaces; *multiplexers*, that forward messages received on their input interfaces to their single output interface ; *aggregators*, that have one or several input interfaces to receive the messages to be aggregated, and one output interface on which to deliver the aggregated message; *deaggregators*, that are dual to aggregators; *conduits*, that allow messages to be exchanged between different address spaces.

Figure 1[1] shows a simple assemblage of Dream components that corresponds to two communicating sites, `Site A` sending different kinds of messages to `Site B`. The assemblage is constituted by two generator components, `Gen1` and `Gen2`, that emit different messages. These messages are then sent to a multiplexer, to be handled by the TCP/IP conduit, and transferred to `Site B`. On `Site B`, router `R` forwards messages to the `Handler 1` or `Handler 2` component, based on the structure of the incoming messages.

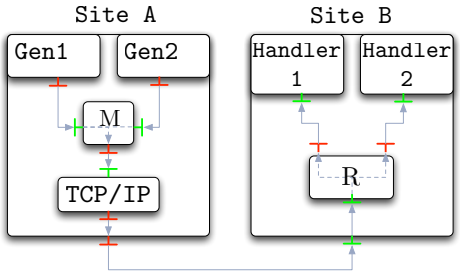

Figure 1. A DREAM Assemblage

Verifying the correctness of the assemblage implies verifying structural constraints to the effect that input and output interfaces are properly matched, but also message manipulation behavior of components, to ensure that a component does not receive a message it is not able to handle. In our simple example above, this could be the case e.g. if the router `R` was not able to discriminate correctly between messages destined to the two handler components. Suppose, for instance, that `Gen1` (resp. `Gen2`) generates messages having the named chunks `a` and `b` (resp. `a` and `c`). Then, if the router `R` routes only on the presence of the field `a`, the two kinds of messages (which are multiplexed to the same conduit) are not distinguished by the router, and are incorrectly sent to the same handler `Handler1`. In the presence of complex assemblages, such an analysis can quickly become difficult.

[1] The figure is simplified for brevity, e.g. one would expect a dual TCP/IP component to complete the conduit on Site B.

3. Calculus

We define in this section a process calculus intended to capture the operational essence of Dream components. The level of abstraction of the calculus is similar to that of an architecture description language, and thus more amenable to formal analysis than if we were working at the level of the host programming language (Java). The calculus is a sub-calculus of the Kell calculus [34], where *localities* correspond to components, and interfaces are modelled by *channels* similar to π-calculus channels. Channels carry *extensible records*, which model messages. Message chunks are modelled by record fields, while message manipulation is modelled in our calculus by operations on records taken from [33].

For the sake of simplicity, instead of supporting full pattern matching on channel payload as in the Kell calculus, we include a special feature to directly encode a router-like behavior: $\mathrm{IfPre}(a, M, i, s)$. This construct tests if the chunk named 'a' is present in the message 'M'. If it is, the message is sent on the 'i' output channel (i.e. interface) , and on the 's' output channel if the chunk is absent.

Abstract syntax The syntax of our calculus is given by the following grammar:

$$
\begin{array}{rcl}
D & ::= & b[D] \mid B \mid (D_1 \mid D_2) \\
B & ::= & 0 \mid R \mid e(x).(B_1 \mid \ldots \mid B_n) \mid !B \\
R & ::= & \overline{e}\langle M\rangle \mid \mathrm{IfPre}(a, M, s_1, s_2) \\
M & ::= & x \mid \{a_1 = M_1; \ldots; a_n = M_n\} \mid c \mid (M_1\ M_2) \\
c & ::= & .a \mid +_a \mid -_a \mid \ldots
\end{array}
$$

A component $b[D]$ is a locality with name b and content D. $D_1 \mid D_2$ denotes the parallel composition of processes D_1 and D_2. Basic processes B comprise the null process 0, message actions R, receivers, and replications of basic processes. A receiver $e(x).(B_1 \mid \ldots \mid B_n)$ awaits messages on channel e and reacts to the receipt of a message on e by launching in parallel the n processes $B_1 \ldots B_n$. As in the π-calculus, $!B$ stands for the replication of process B. In the following, e and s range over channel names. A message action R is either the sending of a message M on a channel e, written $\overline{e}\langle M\rangle$, or a routing process $\mathrm{IfPre}(a, M, s_1, s_2)$. The latter tests whether the message M has a field with label 'a'; if so it sends it on channel s_1; otherwise, it sends it on s_2. Messages comprise variables x, records $\{a_1 = M_1; \ldots; a_n = M_n\}$, constants c, and applications $(M_1\ M_2)$. Constants allow us to parameterize the calculus over a set of basic values and data type operations. By definition, constants include at least record operations: field selection $.a$, field addition $+_a$, and field removal $-_a$.

Examples An example *multiplexer* can be defined as

$$\texttt{Mult} \triangleq b[!e_1(x).\overline{s}\langle x\rangle \mid !e_2(x).\overline{s}\langle x\rangle]$$

The two receivers in `Mult`, listening on e_1 and e_2, send messages they receive on output s, thus *multiplexing* them on one output channel.

An example *router* can be defined as

$$\texttt{Router} = b\,[!e(x).\mathrm{IfPre}(a, x, s_1, s_2)]$$

It has three ports: one input (e) and two outputs (s_1 and s_2). The routing behavior in this case is simply implemented using the `IfPre` operator. If the message received on the input contains a field labeled a, then it is sent on s_1, otherwise, it is sent on s_2.

An example *binding* or *connector* can be defined as

$$\texttt{Conn} \triangleq !s(x).\overline{e}\langle x\rangle$$

Assume component b_1 can send messages on port s, and component b_2 can receive messages on port e: in the assemblage $b_1[..] \mid \texttt{Conn} \mid b_2[...]$, the connector `Conn` allows b_1 and b_2 to communicate by forwarding messages between them.

Figure 2 presents an encoding of the `Site A` assemblage shown in Figure 1 (in the figure, horizontal superposition inside component brackets [...] implies parallel composition). Generators send

$$A \left[\begin{array}{lcl} \texttt{Gen1}[!\overline{g_1}\langle\{a=1;b=1\}\rangle] & \mid & !g_1(x).\overline{m_1}\langle x\rangle \\ \texttt{Gen2}[!\overline{g_2}\langle\{a=1;c=1\}\rangle] & \mid & !g_2(x).\overline{m_2}\langle x\rangle \\ \texttt{M}[!m_1(x).\overline{m_o}\langle x\rangle \mid !m_2(x).\overline{m_o}\langle x\rangle] & \mid & !m_o(x).\overline{t_i}\langle x\rangle \\ \texttt{TCP/IP}\left[!t_i(x).\overline{t_o}\langle\left\{\begin{array}{c} ip=\ldots; \\ val=x \end{array}\right\}\rangle\right] & \mid & !t_o(x).\overline{o}\langle x\rangle \end{array}\right]$$

Figure 2. Site A

messages on their output channel, 'g_1' for `Gen1` and 'g_2' for `Gen2`. `Gen1` sends messages with two chunk named 'a' and 'b', while `Gen2` sends messages having 'a' and 'c' chunks. Bindings are encoded as shown in the preceding example, using a simple forwarding function, as in $!g_1(x).\overline{m_1}\langle x\rangle$ which connects the output channel of `Gen1` to the first input channel of the multiplexer. Finally, the `TCP/IP` component sends messages containing two chunks: 'ip' contains an IP adress, and 'val' contains the message to transmit to Site B. The output channel of this component is then bound to the channel 'o', which is the output channel of Site A.

Operational Semantics The operational semantics of our calculus is defined by a reduction relation, denoted $\triangleright$, defined on closed terms. It is defined modulo a structural equivalence relation on process terms, and evaluation contexts. Informally, the structural equivalence makes the parallel operator commutative, associative, with neutral element 0, allows the unfolding of replications, and makes irrelevant the order of fields in a record. For brevity, we do not give the formal definition of the structural equivalence here.

The reduction relation is defined as the smallest relation that verifies the rules of Figure 3. The two context rules mean that reduction, i.e. execution, can happen anywhere in the program. Evaluation contexts are terms with a *hole* [], given by the following grammar:

$$E ::= [] \quad | \quad \{a_1 = E; a_2 = M_2; \ldots; a_n = M_n\} \quad | \quad (M\ E)$$
$$| \quad (E\ M) \quad | \quad \overline{s}\langle E\rangle \quad | \quad \text{IfPre}(a, E, s_1, s_2) \quad | \quad (E \mid D) \quad | \quad b[E]$$

The rules SELECT, ADD, and REMOVE correspond to record operations (field selection, field addition, and field removal, respectively). They can be complemented by additional rules of application for constants (not shown here).

We have two possible reductions for the 'IfPre' construct: when the given message 'M' has the 'a' chunk, it is sent on the first output channel 's_1' (rule IFPRE); otherwise ('M' does not have the 'a' chunk), the message is sent on the second output channel 's_2' (rule IFABS). Finally, we have three communication rules: COM1 allows communication between two programs in a same component; COM2 allows reception of messages on an input interfaces; COM3 allows emission on output interfaces of a component.

CONTEXT-D
$$\frac{D \triangleright D'}{E[D] \triangleright E[D']}$$

CONTEXT-M
$$\frac{M \triangleright M'}{E[M] \triangleright E[M']}$$

SELECT
$$\frac{M = \{a = M_1; a_2 = M_2; \ldots; a_n = M_n\}}{(.a\ M) \triangleright M_1}$$

ADD
$$\frac{M = \{a_1 = M_1; \ldots; a_n = M_n\} \qquad \forall 0 < i \leq n,\ a_i \neq a}{((+_a\ M)\ M') \triangleright \{a = M'; a_1 = M_1; \ldots; a_n = M_n\}}$$

REMOVE
$$\frac{M = \{a = M_1; a_2 = M_2; \ldots; a_n = M_n\}}{(-_a\ M) \triangleright \{a_2 = M_2; \ldots; a_n = M_n\}}$$

IFPRE
$$\frac{M = \{a = M_1; a_2 = M_2; \ldots; a_n = M_n\}}{\text{IfPre}(a, M, s_1, s_2) \triangleright \overline{s_1}\langle M\rangle}$$

IFABS
$$\frac{M = \{a_1 = M_1; \ldots; a_n = M_n\} \qquad \forall 0 < i \leq n,\ a_i \neq a}{\text{IfPre}(a, M, s_1, s_2) \triangleright \overline{s_2}\langle M\rangle}$$

COM1
$$\overline{e}\langle M\rangle \mid e(x).(B_1 \mid \ldots \mid B_n) \triangleright (B_1 \mid \ldots \mid B_n)\{^M/_x\}$$

COM2
$$\overline{e}\langle M\rangle \mid b[e(x).(B_1 \mid \ldots \mid B_n) \mid D] \triangleright b[(B_1 \mid \ldots \mid B_n)\{^M/_x\} \mid D]$$

COM3
$$b[\overline{e}\langle M\rangle \mid D] \mid e(x).(B_1 \mid \ldots \mid B_n) \triangleright b[D] \mid (B_1 \mid \ldots \mid B_n)\{^M/_x\}$$

Figure 3. Reduction rules

Message Errors We define in this section the class of errors that our type system is intended to avoid. Our definition is based on the notion of *value*, and catches message manipulation errors, and routing errors, i.e. when the input message of a router is not a valid message. A value is a message which cannot be reduced. Values are given by the following grammar:

$$v \quad ::= \quad \{a_1 = v_1; \ldots; a_n = v_n\} \quad | \quad x \quad | \quad c$$

DEFINITION 3.0.1. *A program D has a* Message Error *iff either:*

- *There exist E, v, and v' such that $D = E[(v\ v')]$ and $(v, v') \notin$* `match`.
- *There exist E, v, a, s_1, and s_2 such that $D = E[\text{IfPre}(a, v, s_1, s_2)]$ with v not being a record.*

Relation `match` is defined to capture application errors. It is parameterized by the set of constants used. By definition, it includes at least the pairs (c, M), where c denotes a record operation, and M is a message meeting the premise predicate in the rules ADD, REMOVE, and SELECT. For instance, the operation $.a$ may only be applied to records containing a field a. Hence the only pairs of the form $(.a, M)$ that relation `match` contains are those where M is a message containing an a field.

Example Figure 4 presents an encoding of the `Site B` from the assemblage described Figure 1. As in Figure 1, this component must be read from bottom to top: the input channel of Site B is 'i', which is bound to the input of the router component '`R`'. The router extracts the '*value*' chunk from its input messages, and sends the result depending on the presence of the 'a' chunk on the output channel 's_1' or 's_2'. 's_1' (resp. 's_2') is bound to the input interface of `Handler1` (resp. `Handler2`). `Handler1` retrieves the

$$B \left[\begin{array}{l} \texttt{Handler1}[!h_1(x).\overline{hi_1}\langle x.c\rangle] \mid \texttt{Handler2}[!h_2(x).0] \\ !s_1(x).\overline{h_1}\langle x\rangle \mid \texttt{R}[!r(x).\text{IfPre}(a, x.val, s_1, s_2)] \mid !s_2(x).\overline{h_2}\langle x\rangle \\ !i(x).\overline{r}\langle x\rangle \end{array} \right]$$

Figure 4. Site B

T	$::=$	$E \mid \forall\alpha.T$	Base type
E	$::=$	$E \rightarrow E' \mid \{W^{\emptyset}\} \mid \eta \mid \text{int}$	Element type
W^l	$::=$	$a : K; W^{l \uplus \{a\}} \mid \text{Abs}^l \mid \rho^l$	Rows
K	$::=$	$\text{Pre}(E) \mid \text{Abs}$	Presence
S	$::=$	$\emptyset \mid e \mid e : (T) \mid S \cup S'$	Process type
τ	$::=$	$E \mid S$	Any type

'c' chunk of its input messages, while `Handler2` throws away its input messages.

Site B alone has no error, but the assemblage of Site A and Site B gives rise to one: the message $\{a = 1; b = 1\}$ will be given as input to `Handler1`, which cannot access to the undefined 'c' chunk.

4. Type system

This section describes our type system, which is designed to ensure that no Message Error can occur during the execution of a well-typed component. Our type system is based in the idea presented in [6]: using *rows* [33] to check the validity of message manipulation. We add *process types* [39, 22] to be able to type components. We also use two specific typing rules inspired by *intensional type analysis* [11] to handle the routing procedure.

Type Syntax. The syntax of our types is given by the grammar below. The set $\mathcal{V}^m$ consists of *base type variables*, ranged over by η and its variants. The sets $\mathcal{V}^l$, where l is a finite set of labels, consist of *row variables* whose instantiation cannot contain any label in the set l. They are ranged over by ρ^l and their variants. The set $\mathcal{V}^k$ is used for the field presence syntactic definition. Its variables are ranged over by κ and its variants. Finally, the set $\mathcal{V}^s$ consists of *variables for process types*, ranged over by ζ and its variants. When no precision is needed, we write α or β a type variable of any of these sets, and $\mathcal{V}$ the set of all such variables. Base types are used to type messages. They comprise type variables for polymorphism, *rows*, functions and basic types constructors $t(E_1, \ldots, E_n)$ for dealing with constants which are not record operations (they can comprise e.g. nullary constructors corresponding to primitive types such as int and string). *Rows* are used to type messages: a row defines which chunks are present in a typed message, and what it contains. For instance, the message $\{a = 1; c = \text{“}\texttt{record}\text{”}\}$ is typed $\{a : \text{Pre(int)}; c : \text{Pre(string)}; \text{Abs}^{\{a,c\}}\}$: chunks named a and c are present and respectively contain an integer and a string. Informally, a row consists of a list of chunk names, defining for each name if its corresponding chunk is present or not. This list can end either with Abs, meaning that all other chunks are absent from the message, or a *row variable* ρ^l for polymorphism. Finally, the kind 'l' on row definition W^l is a set of chunk names and ensures that a chunk is at most defined once (W^l denotes the set of rows whose field (or chunk) names do not belong to l).

Process types are used to type components. They declare the channels used by a program, and map them to finite, possibly empty, sets of base types. Such sets of base types correspond to the type of messages channels can transmit during the program execution. For instance, the type $e : (\text{int})$ states that an integer can be transmitted over channel e. The type $\emptyset$ states that no channel is used by the typed process. The type e declares channel e, and maps it to an empty base type set, whereas $e : (T)$ declare e, and maps it to the singleton $\{T\}$. The union construct $S \cup S'$ declares all the channels in S and S', and maps them to the union of the mapping induced by S and S'. For instance, the type $s \cup e : (T_1) \cup e : (T_2)$ declares the channels s and e, maps s to the empty set, and e to $\{T_1, T_2\}$. In the following we write $dc(S)$ the set of all channels declared by S, and $S(e)$ the set it maps to e.

4.1 Typing Messages and Components

Types are associated to messages and components using some *typing rules*. Our rules are defined modulo a structural equivalence on types and substitutions. Informally, structural equivalence identifies types with the same meaning, e.g. states that the union is associative, commutative, with a neutral element $\emptyset$, and the order of field definitions in a row is irrelevant. A substitution σ is a simple finite mapping from type variables to types. We note $\text{dom}(\sigma)$ its domain.

Our typing rules are presented Figures 5 and 6, which respectively handle typing messages and typing components. Typing rules are given modulo a structural equivalence between types (not given here for brevity), that essentially states that the union is associative, commutative, with a neutral element $\emptyset$, and the order of field definitions in a row is irrelevant. Type judgments have the form $\Gamma \vdash \texttt{L} : \tau$, where Γ is a typing environment, $\texttt{L}$ is the typed construct, i.e. either a message M or a component D, and τ its type. We use '*fv*' to denote the free variables of a type.

T:VAR
$$\Gamma \vdash x : \Gamma(x)$$

T:SELECT
$$\Gamma \vdash .a : \forall\eta, \rho^{\{a\}}.\{a : \text{Pre}(\eta); \rho^{\{a\}}\} \rightarrow \eta$$

T:ADD
$$\Gamma \vdash +_a : \forall\eta, \rho^{\{a\}}.\{a : \text{Abs}; \rho^{\{a\}}\} \rightarrow \eta \rightarrow \{a : \text{Pre}(\eta); \rho^{\{a\}}\}$$

T:REMOVE
$$\Gamma \vdash -_a : \forall\eta, \rho^{\{a\}}.\{a : \text{Pre}(\eta); \rho^{\{a\}}\} \rightarrow \{a : \text{Abs}; \rho^{\{a\}}\}$$

T:MESSAGE
$$\frac{\forall 0 < i \leq n, \Gamma \vdash M_i : E_i}{\Gamma \vdash \{a_1 = M_1; ..; a_n = M_n\} : \{a_1 : \text{Pre}(E_1); ..; a_n : \text{Pre}(E_n); \text{Abs}\}}$$

T:APP
$$\frac{\Gamma \vdash M_1 : E \rightarrow E' \qquad \Gamma \vdash M_2 : E}{\Gamma \vdash (M_1\ M_2) : E'}$$

T:INST
$$\frac{\Gamma \vdash M : \forall\alpha.T \qquad \text{dom}(\sigma) = \{\alpha\}}{\Gamma \vdash M : \sigma(T)}$$

T:GEN
$$\frac{\Gamma \vdash M : T \qquad \alpha \in fv(T) \setminus fv(\Gamma)}{\Gamma \vdash M : \forall\alpha.T}$$

Figure 5. Typing rules for messages

Typing rules for messages are inspired by [31]. The first rule defines the type of calculus variables from the typing environment Γ. The next three rules present the types of operations on record. For brevity, we do not include rules pertaining to constants and basic type constructors. The rule T:MESSAGE states that we keep track of every chunks inside a message. T:APP simply states that given a valid input, an operator will gives the expected output. The last two rules correspond to type instantiation and type generalization. Note that α stands for either base type or row variables.

The typing rules for processes can be seen as a set of constraints imposed on process types: this allows flexibility without requiring a *sub-typing* rule for processes as in [40]. The rule T:CHANNEL

$$\text{T:CHANNEL}\quad \frac{\Gamma \vdash M : T \qquad T \in S(s)}{\Gamma \vdash \overline{s}\langle M\rangle : S}$$

$$\text{T:IFPRE1}\quad \frac{\Gamma \vdash M : T \qquad T = \forall\overline{\alpha}.\{a : \text{Pre}(\ldots); \ldots\} \qquad T \in S(s_1)}{\Gamma \vdash \text{IfPre}(a, M, s_1, s_2) : S}$$

$$\text{T:IFPRE2}\quad \frac{\Gamma \vdash M : T \qquad T = \forall\overline{\alpha}.\{a : \text{Abs}; \ldots\} \qquad T \in S(s_2)}{\Gamma \vdash \text{IfPre}(a, M, s_1, s_2) : S}$$

$$\text{T:RECEIVER}\quad \frac{e \in dc(S) \qquad \forall T \in S(e),\ \forall 1 \leq i \leq n,\ (\Gamma \uplus \{x : T\} \vdash B_i : S)}{\Gamma \vdash e(x).(B_1 \mid \ldots \mid B_n) : S}$$

$$\text{T:PARALLEL}\quad \frac{\Gamma \vdash D_1 : S \qquad \Gamma \vdash D_2 : S}{\Gamma \vdash D_1 \mid D_2 : S} \qquad \text{T:ZERO}\quad \Gamma \vdash 0 : S \qquad \text{T:BANG}\quad \frac{\Gamma \vdash B : S}{\Gamma \vdash !B : S}$$

$$\text{T:BOX}\quad \frac{\Gamma \vdash D : S}{\Gamma \vdash b[D] : S}$$

Figure 6. Typing rules for processes

requests that a type S for $\overline{s}\langle M\rangle$ *must* define the channel 's' and map it to a set containing T: for instance, $s : (T) \cup s : (T') \cup e$ is a valid type for $\overline{s}\langle M\rangle$. We have two typing rule to handle the 'IfPre' construct. T:IFPRE1 is applied when the input message is typed with a type of the form $\forall\overline{\alpha}.\{a : \text{Pre}(\ldots); \ldots\}$: it contains a chunk named 'a'. This message will be sent on 's_1', which is represented in the typing rule by the constraint $T \in S(s_1)$. T:IFPRE2 is applied in the other case, i.e. when the input message does not contain 'a': the message is sent on s_2, which is taken in account by the rule via the constraint $T \in S(s_2)$. The rule T:RECEIVER allows for a receiver to accept as input messages with different structures. Indeed, $S(e)$ can contain different types corresponding to messages with different structures, and each of these types are checked independently against the 'B_i's. The combination of this rule and the IFPRE rules are the source of the expressive power of the router construct.

Finally, process types essentially ignore the hierarchical structure of components. This is a simplification which can be easily lifted if one wants to faithfully reflect in process types the encapsulation that components realize.

Typing examples. *Multiplexer.* The multiplexer `Mult` introduced in Section 3 can admit several types, depending on the messages it has in input. For instance, $e_1 \cup e_2$ and $e_1 : (T) \cup e_2 : (T_1) \cup e_2 : (T_2) \cup s : (T) \cup s : (T_1) \cup s : (T_2)$ are valid types for this program. One can for instance verify the second type using the typing rules T:PARALLEL and T:RECEIVER on the processes $e_1(x).\overline{s}\langle x\rangle$ with x typed T, and $e_2(x).\overline{s}\langle x\rangle$ with x first typed T_1 and then typed T_2.

Router. It may seem that the typing rules are too restrictive to capture the expressiveness of the 'IfPre' construct. However, consider what happens when typing a router program such as $e(x).\text{IfPre}(a, x, s_1, s_2)$. Because of rule T:RECEIVER, we have to consider all the message types mapped on channel e. Combining the rules T:IFPRE1, T:IFPRE2 and T:RECEIVER, we can thus recover expected types for the routing process. For instance, one can verify that the following process type can be derived for the router program above, using the rules T:IFPRE1, T:IFPRE2 and T:RECEIVER:

$$\begin{aligned} S \triangleq\ & e : (\{a : \text{Pre(int)}; \text{Abs}^{\{a\}}\}) \\ & \cup e : (\{b : \text{Pre(int)}; \text{Abs}^{\{b\}}\}) \cup e : (\{\text{Abs}^{\emptyset}\}) \\ & \cup s_1 : (\{a : \text{Pre(int)}; \text{Abs}^{\{a\}}\}) \\ & \cup s_2 : (\{b : \text{Pre(int)}; \text{Abs}^{\{a\}}\}) \cup s_2 : (\{\text{Abs}^{\emptyset}\}) \end{aligned}$$

$$\begin{aligned} & g_1 : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ g_2 : (\{a : \text{Pre(int)}; c : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ m_1 : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ m_2 : (\{a : \text{Pre(int)}; c : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ m_o : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ m_o : (\{a : \text{Pre(int)}; c : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ t_i : (\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ t_i : (\{a : \text{Pre(int)}; c : \text{Pre(int)}; \text{Abs}\}) \\ & \cup\ t_o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}); \text{Abs}\}) \\ & \cup\ t_o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; c : \text{Pre(int)}; \text{Abs}\}); \text{Abs}\}) \\ & \cup\ o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; b : \text{Pre(int)}; \text{Abs}\}); \text{Abs}\}) \\ & \cup\ o : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre(int)}; c : \text{Pre(int)}; \text{Abs}\}); \text{Abs}\}) \end{aligned}$$

Figure 7. Type of Site A

$$\begin{aligned} & i : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre}(\eta_1); c : \text{Pre}(\eta_2); \rho\}); \text{Abs}\}) \\ & \cup i : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Abs}; \rho^{\{a\}}\}); \text{Abs}\}) \\ & \cup r : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Pre}(\eta_1); c : \text{Pre}(\eta_2); \rho\}); \text{Abs}\}) \\ & \cup r : (\{ip : \text{Pre(IP)}; val : \text{Pre}(\{a : \text{Abs}; \rho^{\{a\}}\}); \text{Abs}\}) \\ & \cup s_1 : (\{a : \text{Pre}(\eta_1); c : \text{Pre}(\eta_2); \rho^{\{a,c\}}\}) \\ & \cup s_2 : (\{a : \text{Abs}; \rho^{\{a\}}\}) \\ & \cup h_1 : (\{a : \text{Pre}(\eta_1); c : \text{Pre}(\eta_2); \rho^{\{a,c\}}\}) \\ & \cup hi_1 : (\eta_2) \\ & \cup h_2 : (\{a : \text{Abs}; \rho^{\{a\}}\}) \end{aligned}$$

Figure 8. Type of Site B

Simple assemblage. As previously stated, Site A and Site B from Figure 1 are typable, and indeed, we can find a type for each of these components, as presented in Figures 7 and 8. In these types, we omitted to write the kind of the row 'Abs' for more readability. Because our type system keeps track of every messages in every channels, the types can become quite large. However, they are easy to understand. Let's take for instance the type of Site A (Figure 7). The 1st line states that `Gen1` generates messages having a 'a' and a 'b' chunk, each containing an integer. The 3rd line states that these messages are transmitted to the input channels of the multiplexer, which will send them on its output interface (line 5). The 7th and 8th lines state that the messages are transmitted to the input channel of the `TCP/IP` component. The output of the `TCP/IP` component is then described line 9 and 10, and transmitted to the output of Site A (line 11 and 12).

The type for Site B can be read likewise, from its input channel to the handler components. Note that Site B routes all messages with the a chunk through channel s_1, which expects messages with both the a and c chunks. Because of this, connecting Site A and Site B would result in an ill-typed assemblage, and is thus forbidden by our type system.

Type Expressivity. It is interesting to remark that most valid assemblages using a router cannot be well-typed using an ML-like type system. For instance, consider the valid assemblage presented in figure 9. In this example, the component `IP_Protocol` uses an IP address, and the component `FSR_Protocol` uses a time stamp TS. Output messages are sent on the channel o, and then routed depending on the protocol they use: a message having the `IP` field will be sent using the `IP_Protocol` components, and one which doesn't have it is then supposed to be sent using the `FSR_Protocol`

$$\texttt{Producer}\left[\begin{array}{l} !\overline{o}\langle\{\texttt{IP} = 192.168.0.29; \texttt{Val} = \ldots\}\rangle \\ \mid !\overline{o}\langle\{\texttt{TS} = 30; \texttt{Val} = \ldots\}\rangle \end{array}\right]$$
$$\mid !o(x).\overline{r}\langle x\rangle \mid \texttt{Router}[!r(x).\text{IfPre}(\texttt{IP}, x, o_1, o_2)]$$
$$\mid !o_1(x).\overline{ip}\langle x\rangle \mid \texttt{IP_Protocol}[ip(x).(\overline{\text{intern}}\langle x.\texttt{IP}\rangle \mid \ldots)]$$
$$\mid !o_2(x).\overline{fsr}\langle x\rangle \mid \texttt{FSR_Protocol}[fsr(x).(\overline{\text{intern}}\langle x.\texttt{TS}\rangle \mid \ldots)]$$

Figure 9. A simple valid Assemblage

component. In our example, we have a producer component, named `Producer` which send messages for both protocols.

This assemblage can be typed using our type system with the process type presented Figure 10, but has no valid type using e.g. guarded algebraic data types.

$$\begin{array}{l} o : (\{\texttt{IP} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, o : (\{\texttt{TS} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, r : (\{\texttt{IP} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, r : (\{\texttt{TS} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, o_1 : (\{\texttt{IP} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, o_2 : (\{\texttt{TS} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, ip : (\{\texttt{IP} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, ts : (\{\texttt{TS} : \text{Pre(int)}; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}\}) \\ \cup\, intern : (\text{int}) \end{array}$$

Figure 10. A type for the simple valid Assemblage

Informally, messages sent on o have the fields `Val` and either `IP` or `TS`. Thus, using row polymorphism and sub-typing, we can characterize these messages at most with a type of the form $\{\texttt{IP} : \bot; \texttt{TS} : \bot; \texttt{Val} : \text{Pre}(\ldots); \text{Abs}^{\{\texttt{IP},\texttt{TS},\texttt{Val}\}}\}$. But such an input type is not enough for the `IP_Protocol` component, which requires messages with the field `IP` defined. For the same reason, such an assemblage cannot be well-typed in PICT.

4.2 Type System Properties

This type system is sound with respect to message errors, as stated by the correction and subject reduction theorems. The proof of these theorems is classical, and can be found in the companion technical report [19]. In the following, `L` stands for either a message M or a component D, and τ stands for either a base type T or a process type S.

THEOREM 4.2.1 (Correction). *Given a valid typing statement* $\emptyset \vdash \texttt{L} : \tau$, `L` *has no Message Error.*

THEOREM 4.2.2 (Subject reduction). *Given a valid typing statement* $\Gamma \vdash \texttt{L} : \tau$ *and a valid reduction* $\texttt{L} \rhd \texttt{L}'$, *there exists a type derivation of* $\Gamma \vdash \texttt{L}' : \tau$.

Finally, using an encoding of the Post Correspondence Problem [28], we also have the following theorem (whose proof can also be found in the companion technical report [19]):

THEOREM 4.2.3. *Type inference is undecidable.*

This result is closely related to the following one:

FACT 1. *The type system doesn't have the principal type property.*

The notion of principal type in our setting is defined as follows (where $dc(S)$ is the set of channels that are defined in S):

DEFINITION 4.2.1. *Let suppose given a program D. A process type S is a* principal type *for D iff*

- *There exists Γ such that $\Gamma \vdash D : S$ holds;*
- *for all S' with $dc(S') \subset dc(D)$ and Γ' such that $\Gamma' \vdash D : S'$ holds, there exists a substitution σ such that $S' \subset \sigma(S)$.*

Our type system has the principal type property *iff every typable program has a principal type.*

Here is a program D which doesn't have a principal type:

$$D \triangleq b[!e(x).\text{IfPre}(a, x, i, s) \mid i(x).\overline{e}\langle(x.a)\rangle]$$

This component returns the record which appears in a chunk located at the bottom of an arbitrary deep hierarchy of a chunks. For instance, if one send $\{a = \{\}\}$ (resp. $\{a = \{a = \{b = 2\}\}; c = 3\}$) on the channel e, the component will return on the channel s the message $\{\}$ (resp. $\{b = 2\}$). Such a component admits valid types, such as $e \cup i \cup s$. However, it admits no principal type.

We give the proof of this here because it provides some insights on the more unusual phenomena in our type system. Suppose there exist a typing environment Γ and a process type S such that $\Gamma \vdash D : S$ holds. We thus have $e \in dc(S)$. Let consider the two cases:

(i) $S(e) = \emptyset$: we clearly have that S is not principal. Indeed, Let's take $S' = e : (\{\text{Abs}\}) \cup i \cup s : (\{\text{Abs}\})$. It is easy to see that $\Gamma \vdash D : S'$ holds, and there is no substitution σ such that $\{\text{Abs}\} \in \sigma(\emptyset)$.

(ii) Let now suppose $S(e) \neq \emptyset$. We define:

- A family of context: $E_a^{(0)} ::= [\ ]$ $\quad E_a^{(n+1)} ::= \{a : \text{Pre}(E_a^{(n)}); W^{\{a\}}\}$.
- A function d_a on record type schemes such that $d_a(T) = n$ iff there exists a set of variables $\overline{\alpha}$, a context $E_a^{(n)}$ and a type E which has not the form $\{a : \text{Pre}(E'); W^{\{a\}}\}$ such that $T = \forall\overline{\alpha}.E_a^{(n)}[E]$.

Because the channel e is the input channel of a routing procedure, it is evident that all $T \in S(e)$ has the form $\forall\overline{\alpha}.\{W^{\emptyset}\}$.

Let define $n = \max_{T \in S(e)}(d_a(T))$ and take $T \in S(e)$ (we write $T = \forall\overline{\alpha}.E$) such that $d_a(T) = n$. We define $T' = \forall\overline{\alpha}.\{a : \text{Pre}(E); \text{Abs}\}$ and $S' = S \cup e : (T') \cup i : (T')$. Per construction, we have $\vdash D : S'$. As S is a principal type, there exists σ and $T_1 \in S(e)$ such that $T' = \sigma(T_1)$. As we have $d_a(T_1) < d_a(T')$ (by construction), $T_1 = \forall\overline{\alpha}'.E_1$, where E_1 can either be of the form (with $n = d_a(T_1)$):

- $E_a^{(n)}[\{a_1 : K_1; \ldots; a_m : K_m; \rho^l\}]$ with $l = \{a_i \mid 1 \leq i \leq m\}$,$a \notin l$ and $\sigma(\rho) = a : \text{Pre}(E'); W^{l \cup \{a\}}$.
- $E_a^{(n-1)}[\{a : \text{Pre}(\alpha); W^{\{a\}}\}]$ with $\sigma(\alpha) = \{a : \text{Pre}(E'); W^{\{a\}}\}$.

We can remark that if $\forall\overline{\alpha}.\{a : \text{Pre}(E); W^{\{a\}}\} \in S(e)$, then $\forall\overline{\alpha}.E \in S(e)$ (by definition of the type system). Thus, inductively, we can see that $\forall\overline{\alpha}.E_a^{(n)}[E] \in S(e) \Rightarrow \forall\overline{\alpha}.E \in S(e)$. So, we have $\{a_1 : K_1; \ldots; a_m : K_m; \rho^l\}$ or α in $S(e)$, which is impossible: the routing procedure only accepts on input types where the field a is defined (present or absent). Thus S cannot be a principal type for D.

5. Type inference

We present in this section an inference semi-algorithm, i.e. an algorithm which computes a type for typable programs, but doesn't terminate for all inputs. The semi-algorithm is constructed in two steps: a total type inference algorithm for R constructs (message actions), and the semi-algorithm proper, called the propagation algorithm. The resulting typing is not *principal*, but *minimal*. The notion of minimal type is defined below.

DEFINITION 5.0.2. *Given a typing environment* Γ *and two type schemes* T *and* T'*,* T' *is* derived *from* T *for* Γ *(written* $\Gamma : T' \Leftarrow T$*) iff:*

- *there exists a set of type variables* $\overline{\alpha}$ *with* $\overline{\alpha} \cap fv(\Gamma) = \emptyset$ *and a type* T_1 *such that* $T = \forall\overline{\alpha}.T_1$*;*
- *there exists a set of type variables* $\overline{\gamma}$ *with* $\overline{\gamma} \cap fv(\Gamma) = \emptyset$ *and a type* T_2 *such that* $T' = \forall\overline{\gamma}.T_2$*; and*
- *there exists a substitution* σ *with* $\mathrm{dom}(\sigma) \subset \overline{\alpha}$ *and* $\sigma(T_1) = T_2$*.*

Informally, $\Gamma \vdash T_2 \Leftarrow T_1$ means that T_1 is *more general* than T_2, i.e. any message typed T_1 can be typed with T_2. This definition is a bit technical, but it allows a simple manipulation of type schemes. Here is a simple example involving row manipulation: $\Gamma \vdash \{a : \mathrm{Pre}(\mathrm{int} \to \mathrm{int}); \mathrm{Abs}\} \Leftarrow \forall\eta_1, \rho.\{a : \mathrm{Pre}(\eta_1); \rho\}$.

DEFINITION 5.0.3. *Given a typing environment* Γ *and two process types* S_1*,* S_2*, we say that* S_2 *is* derived *from* S_1 *for* Γ*, written* $\Gamma : S_2 \Leftarrow S_1$*) iff*

- $dc(S_1) \subset dc(S_2)$*.*
- *For all* $e \in dc(S_1)$ *and all* $T \in S_1(e)$*, there exists* $T' \in S_2(e)$ *with* $\Gamma : T' \Leftarrow T$*.*

DEFINITION 5.0.4. *Given a typing environment* Γ *and a program* D*, we say that* D *admits a* minimal type S *for* Γ *iff:*

- $\Gamma \vdash D : S$ *holds.*
- *For all* S' *such that* $\Gamma \vdash D : S'$*, we have* $\Gamma : S' \Leftarrow S$*.*
- $fv(S) \setminus fv(\Gamma) = \emptyset$*.*

We finally note $\Gamma \vdash_m D : S$ *when* S *is a minimal type for* D*,* Γ*.*

Note that a minimal type is "minimal" in terms of process type inclusion, while it is the most "general" in terms of message types. Note also that the last condition in the definition ensures that a minimal type is fully *generalized*.

5.1 The inference for R constructs

The inference algorithm for message actions is *constraint-based* [2, 26, 29], and thus, works in two phases.

In the *constraint generation phase*, our algorithm explores the structure of the program, in order to extract constraints the types must verify. Basic functions have generic types which are instantiated, for instance before the typing rule T:APP, to see if the parameter is valid, and to define the type of the result of the application. This instantiation is encoded via an *equality constraint*: given the type annotations $c : \forall\overline{\alpha}.E \to E'$ and $M : E''$, the application $(c\ M)$ raises the constraint $E = E''$, meaning that the substitution we compute must unify E with E''.

A second kind of constraint is needed by the routing procedure. Indeed, as this step of the inference doesn't compute types – only constraints – we cannot know the structure of a message M. Thus, it is impossible for the inference algorithm, to know on which channel the program $\mathrm{IfPre}(a, M, s_1, s_2)$ will send the message M. The only way the constraint generation algorithm has to represent the type of such program is as a process type variable. Moreover, the instantiation of this variable is defined by the presence or not of the field a in the message M. The constraints we use to encode such a conditional instantiation are *conditional constraints* [2, 29]. Such constraints make use of K variables to represent the unknown state of the field the routing procedure is based on.

The purpose of the *constraint resolution phase* is to compute a substitution satisfying the constraints generated by the constraint generation phase. Applying this substitution to the process type computed by the constraint generation phase results in a valid type for the program. Specifically, a constraint can be viewed as a finite set of equality and conditional constraints. We can easily define a substitution validating an equality constraint. By combining these different substitutions, we then have enough knowledge of the types of the different messages to compute the substitutions standing for the routing procedures. We can then compute the substitution corresponding to all the *simple* constraints in the set, and combine them to obtain the wanted substitution.

Constraints The set of *simple* constraints is defined by the following grammar:

C_s	::=		*Simple* constraint
		$E = E' \quad \mid \quad K = K'$	Equality constraint
	$\mid$	$K = \mathrm{Pre}?\ S = S'$	Present conditional constraint
	$\mid$	$K = \mathrm{Abs}?\ S = S'$	Absent conditional constraint

A conditional constraint of the form $K = \mathrm{Pre}?\ S = S'$ means that if K is present (there exists E such that $K = \mathrm{Pre}(E)$), then the process types S and S' must be equal. Constraints of the form $K = \mathrm{Abs}?\ S = S'$ are interpreted similarly.

A constraint is a finite conjunction (or finite set) of *simple* constraints, as indicated by the following grammar:

C	::=		Constraint
		true	Empty constraint
	$\mid$	C_s	*Simple* constraint
	$\mid$	$C \wedge C$	Constraint conjunction

DEFINITION 5.1.1. *A substitution* σ satisfies *the simple constraint* C_s *(written* $\sigma \vDash C_s$*) iff either:*

- $C_s = (E = E')$*, and* $\sigma(E) = \sigma(E')$*.*
- $C_s = (K = K')$*, and* $\sigma(K) = \sigma(K')$*.*
- $C_s = K = \mathrm{Pre}?\ S = S'$*, and* $\sigma(K) = \mathrm{Pre}(E)$ *for some* E *implies that* $\sigma(S) = \sigma(S')$*.*
- $C_s = K = \mathrm{Abs}?\ S = S'$*, and* $\sigma(K) = \mathrm{Abs}$ *implies that* $\sigma(S) = \sigma(S')$*.*

DEFINITION 5.1.2. *A substitution* σ satisfies *the constraint* C *(written* $\sigma \vDash C$*) iff either:*

- $C =$ **true**.
- $C = C_s$*, and* $\sigma \vDash C_s$*.*
- $C = C_1 \wedge C_2$*, and* $\sigma \vDash C_1$ *and* $\sigma \vDash C_2$*.*

A constraint is satisfiable*, written* $\vDash C$*, if there exists* σ *such that* $\sigma \vDash C$*.*

Constraint generation The constraint generation algorithm computes two informations:

1. The *basic* type of the input program. Since we make no computation on types at this stage of the algorithm, this type only describes the basic structure of the actual type of the program.
2. The constraint associated to the computed type. These constraints represents the properties of the computed type. From this constraint, the constraint resolution algorithm gives a substitution, which, applied to the basic type, will return a valid type of the program.

Technically, the constraint generation algorithm is constructed using rules of the form $F, \Gamma \vdash \mathtt{L} : [F']\ \tau \mid C$, where:

- F and F' are finite sets of type variables. They represent (as in [30]) the sets of variables already used by our algorithm. F is the set used before the application of the rule, and F' the set resulting from the application of the rule. They formalize the introduction of *fresh* variables.

- Γ is a typing environment.
- $\mathtt{L}$ is the inferred construct: it is either a M or a R construct.
- τ is the *basic* type of the inferred construct. It is either a message type T or a set type S, depending on the inferred construct.
- C is the generated constraint.

The constraint generation algorithm is defined by the rules given in Figure 11.

I:VAR
$$\frac{\Gamma(x) = \forall(\alpha_i)_{0<i\leq n}.E}{F, \Gamma \vdash x : [F \uplus \{\gamma_i \mid 0 < i \leq n\}]\ E\{^{\gamma_i}/_{\alpha_i}\} \mid \textbf{true}}$$

I:PRIMITIVE
$$\frac{\eth(c) = \forall(\alpha_i)_{0<i\leq n}.E}{F, \Gamma \vdash c : [F \uplus \{\gamma_i \mid 0 < i \leq n\}]\ E\{^{\gamma_i}/_{\alpha_i}\} \mid \textbf{true}}$$

I:DEREF
$$\frac{\Gamma(r) = E}{F, \Gamma \vdash !r : [F]\ E \mid \textbf{true}}$$

I:APP
$$\frac{F, \Gamma \vdash M : [F_1]\ E_1 \mid C_1 \qquad F_1, \Gamma \vdash M' : [F_2]\ E_2 \mid C_2}{F, \Gamma \vdash (M\ M') : [F_2 \uplus \{\alpha\}]\ \alpha \mid C_1 \wedge C_2 \wedge (E_1 = E_2 \rightarrow \alpha)}$$

I:MESSAGE
$$\frac{\forall 0 < i \leq n,\ F_{i-1}, \Gamma \vdash M_i : [F_i]\ E_i \mid C_i \qquad \forall i \neq j,\ a_i \neq a_j}{\begin{array}{c} F_0, \Gamma \vdash \{a_1 = M_1; \ldots; a_n = M_n\} : \\ [F_n]\ \{a_1 : \text{Pre}(E_1); \ldots; a_n : \text{Pre}(E_n); \text{Abs}\} \mid (\bigwedge_{0<i\leq n} C_i) \end{array}}$$

I:CHANNEL
$$\frac{F, \Gamma \vdash M : [F']\ E \mid C}{F, \Gamma \vdash \overline{s}\langle M \rangle : [F']\ s : (E) \mid C}$$

I:IFPRE
$$\frac{F, \Gamma \vdash M : [F_1]\ E \mid C_1}{\begin{array}{c} F, \Gamma \vdash \text{IfPre}(a, M, s_1, s_2) : [F_1 \uplus \{\alpha_1, \alpha_2, \alpha_3\}]\ \alpha_3 \mid \\ C_1 \wedge (E = \{a : \alpha_1; \alpha_2\}) \wedge \begin{pmatrix} \alpha_1 = \text{Pre?}\ \alpha_3 = s_1 : (E)) \\ \alpha_1 = \text{Abs?}\ \alpha_3 = s_2 : (E) \end{pmatrix} \end{array}}$$

Figure 11. Constraint generation

Constraint resolution The constraint resolution algorithm computes a substitution validating its input constraint.

DEFINITION 5.1.3. *Given a substitution σ, we write:*

- $\text{dom}(\sigma)$ *the set* $\{\alpha \mid \alpha \in \mathcal{V} \wedge \sigma(\alpha) \neq \alpha\}$.
- $\Im(\sigma)$ *the set* $\bigcup_{\alpha \in \text{dom}(\sigma)} fv(\sigma(\alpha))$.

DEFINITION 5.1.4. *Let C be a constraint such that $\models C$. We write $[\![C]\!]$ the set $\{\sigma \mid \sigma \models C\}$. We say that $\sigma \in [\![C]\!]$ is an* mgu for C *(denoted by $\sigma = \text{mgu}(C)$) iff for all $\sigma_1 \in [\![C]\!]$, there exists σ'_1 such that $\sigma_1 = \sigma'_1 \circ \sigma$.*

DEFINITION 5.1.5. *Given a substitution σ and a constraint C, we write $\sigma = \text{mgu}^i(C)$ iff*

- $\sigma = \text{mgu}(C)$.
- $\Im(\sigma) \cup \text{dom}(\sigma) \subset fv(C)$.
- $\sigma^2 = \sigma$.

The constraint resolution algorithm is given by rules of the form $C \Rightarrow \sigma$ where C is the input constraint and σ is the computed substitution. The rules defining the constraint resolution algorithm are given Figures 12 and 13.

$$\textbf{true} \Rightarrow \text{id} \qquad \text{Abs} = \text{Abs} \Rightarrow \text{id} \qquad \alpha = \alpha \Rightarrow \text{id}$$

$$\frac{\alpha \notin fv(E)}{\alpha = E \Rightarrow (\alpha \rightarrow E)} \qquad \frac{\alpha \notin fv(K)}{\alpha = K \Rightarrow (\alpha \rightarrow K)}$$

$$\frac{K_1 = K_2 \wedge W_1^l = W_2^l \Rightarrow \sigma}{a = K_1; W_1^l = a = K_2; W_2^l \Rightarrow \sigma} \qquad \frac{W^{\emptyset} \Rightarrow \sigma}{\{W^l\} \Rightarrow \sigma}$$

$$\frac{\bigwedge_{1\leq i\leq n} E_i = E'_i \Rightarrow \sigma}{s(E_1, \ldots, E_n) = s(E'_1, \ldots, E'_n) \Rightarrow \sigma}$$

$$\frac{C \Rightarrow \sigma \qquad \sigma(C') \Rightarrow \sigma'}{C \wedge C' \Rightarrow \sigma' \circ \sigma}$$

Figure 12. Equality constraint resolution

$$\text{Abs} = \text{Abs?}\ \alpha = S \Rightarrow (\alpha \rightarrow S) \qquad \text{Pre} = \text{Abs?}\ S = S' \Rightarrow \text{id}$$

$$\text{Abs} = \text{Pre?}\ S = S' \Rightarrow \text{id} \qquad \text{Pre} = \text{Pre?}\ \alpha = S \Rightarrow (\alpha \rightarrow S)$$

Figure 13. Conditional constraint resolution

From the shape of the rules, it is clear this algorithm computes a substitution. We have:

THEOREM 5.1.1 (Constraint resolution). *Given a valid constraint generation statement $F, \Gamma \vdash \mathtt{L} : [F']\ \tau \mid C$, where $fv(\Gamma) \subset F$, the two following properties are equivalent:*

i) C is satisfiable.
ii) there exists σ such that $C \Rightarrow \sigma$ and $\sigma = \text{mgu}^i(C)$.

Properties of the inference algorithm The constraint generation algorithm computes a *basic* type for the input program, and a constraint defining the inner structure of the computed type. The constraint resolution algorithm constructs a substitution from the constraint, which, applied to the *basic* type, gives an instantiated type for the input program.

This computed type is still not a valid type for the input program. Indeed, for the sake of the constraint generation algorithm, we replaced the type schemes with monomorphic types. In order to re-bind the variables in the computed type, we have a generalization operator, defined as follow.

DEFINITION 5.1.6. *Let suppose given a typing environment Γ. The generalization of a message type T for Γ (noted $\text{Gen}(\Gamma, T)$) is the unique type (modulo α-conversion) $\forall(fv(T) \setminus fv(\Gamma)).T$. The generalization of a process type S for Γ (noted $\text{Gen}(\Gamma, S)$) is defined inductively by the following rules:*

$$\text{Gen}(\Gamma, \emptyset) \triangleq \emptyset \qquad \text{Gen}(\Gamma, \zeta) \triangleq \zeta \qquad \text{Gen}(\Gamma, e) \triangleq e$$

$$\text{Gen}(\Gamma, e : (T)) \triangleq e : (\text{Gen}(\Gamma, T))$$

$$\text{Gen}(\Gamma, S_1 \cup S_2) \triangleq (\text{Gen}(\Gamma, S_1)) \cup (\text{Gen}(\Gamma, S_2))$$

The main properties of our algorithm are given by the following:

THEOREM 5.1.2 (Correction). *Given:*

- *A valid constraint generation* $F, \Gamma \vdash \mathtt{L} : [F']\ \tau \mid C$*, with* $fv(\Gamma) \subset F$*, and* $\models C$*.*
- *A valid constraint resolution* $C \Rightarrow \sigma$*.*

Then, there exists a type derivation of $\sigma(\Gamma) \vdash \mathtt{L} : \mathrm{Gen}(\sigma(\Gamma), \sigma(\tau))$*.*

THEOREM 5.1.3 (Completeness). *Given a valid typing statement* $\Gamma \vdash \mathtt{L} : \tau$*, and a set of type variable* F *such that* $fv(\Gamma) \subset F$*, there exist:*

- *A valid constraint generation statement* $F, \Gamma \vdash \mathtt{L} : [F']\ \tau' \mid C$*, with* $\models C$*.*
- *A valid constraint resolution* $C \Rightarrow \sigma$*.*

Moreover, there exist a permutation σ' *such that:*

- $\sigma' \circ \sigma(\Gamma) = \Gamma$*.*
- $\Gamma \vdash \tau \Leftarrow \mathrm{Gen}(\Gamma, \sigma' \circ \sigma(\tau'))$*.*

COROLLARY 1. *Given* R*,* S *and a typing environment* Γ *such that the typing statement* $\Gamma \vdash R : S$ *is valid, there exists a process type* S' *such that* $\Gamma \vdash_m R : S'$*.*

5.2 Propagation Algorithm

The principle of our type inference semi-algorithm (called propagation algorithm) is based on a property of a program's minimal type. Indeed, a minimal type for a program D maps every channel e *used* by D to the set of the types of all messages which may be sent on e. The purpose of this algorithm is then to compute the types of all messages which can be created during the program execution. This computation is done inductively, by adding to a partial type annotation a channel declaration corresponding to a message which may be sent in the program.

For instance, consider the program

$$D \triangleq \overline{e}\langle\{b = \text{`}\mathtt{hi}\text{'}\}\rangle e(x).\overline{s}\langle x + (a = 1)\rangle$$

and the partial annotation $\emptyset$. This annotation can be augmented to $\emptyset \cup e : (\{b : \mathrm{Pre}(\mathrm{string}); \mathrm{Abs}\})$. To this type, we can also add $s : (\{a : \mathrm{Pre}(\mathrm{int}); b : \mathrm{Pre}(\mathrm{string}); \mathrm{Abs}\})$, which will give us the minimal type of the program. This example presents a possible run of our propagation algorithm. The initial annotation is the empty set, to which we add the type of the different messages present in the program, or which might be created during its execution.

Before presenting the algorithm, we give some preliminary definitions.

DEFINITION 5.2.1. *Given two programs* D*,* D' *and a finite family of pair* $(e_i, x_i)_{1 \leq i \leq n}$*, we say that* $(e_1 : x_1; \ldots; e_n : x_n/D')$ *is a* sub-program *of* D*, and write* $(e_1 : x_1; \ldots; e_n : x_n/D') \subset D$*, iff either:*

- *There exists* E *such that* $D = E[D']$ *and* $n = 0$*. In such case, we will write* $(\emptyset/D') \subset D$*,*
- *There exists* E *and* D'' *such that* $D = E[e_1(x_1).(D'')]$ *and* $(e_2 : x_2; \ldots; e_n : x_n/D') \subset D''$*.*

DEFINITION 5.2.2. *The* input set *of a* D *construct, written* $\mathtt{I}(D)$*, consists of the channels* D *is listening on.*

$$\mathtt{I}(0) = \emptyset \qquad \mathtt{I}(\overline{s}\langle M\rangle) = \emptyset \qquad \mathtt{I}(\mathrm{IfPre}(a, M, s_1, s_2)) = \emptyset$$

$$\mathtt{I}(e(x).(B_1 \mid \ldots \mid B_n)) = \bigcup_{1 \leq i \leq n} \mathtt{I}(B_i) \cup \{e\}$$

$$\mathtt{I}(!B) = \mathtt{I}(B) \qquad \mathtt{I}(D_1 \mid D_2) = \mathtt{I}(D_1) \cup \mathtt{I}(D_2)$$

$$\mathtt{I}(b[D]) = \mathtt{I}(D)$$

Propagation algorithm. Our propagation algorithm manipulates judgements of the form $\Gamma \vdash D : S$, where Γ is a typing environment, D is the program whose type is being inferred, and S is the partial annotation computed so far. One step in the execution of the algorithm takes the form: $\Gamma \vdash D : S \dashrightarrow \Gamma' \vdash D : S'$ where $\Gamma \vdash D : S$ is the initial judgment, and $\Gamma' \vdash D : S'$ is the judgment resulting from the propagation step. The algorithm operation is defined as the transitive closure of the relation $\dashrightarrow$, defined by the two propagation rules presented Figure 14. The first rule adds channel declaration corresponding to message which may be send in the program: it is the main rule of the algorithm. The second one is mainly technical and adds the input channel of receivers, as requested in the T:RECEIVER typing rule.

P:MESSAGE

$$\frac{\begin{array}{c}\exists(e_1 : x_1; \ldots e_n : x_n/R) \subset D \qquad \forall 1 \leq i \leq n,\ \exists T_i \in S(e_i) \\ fv(\Gamma) \cup fv(S), \Gamma; x_1 : T_1; \ldots x_n : T_n \vdash R : [F']\ S' \mid C \\ C \Rightarrow \sigma \qquad A = (\mathrm{Gen}(\sigma(\Gamma), \sigma(S')) \not\subset \sigma(S)) \\ B = (\exists \alpha_1 \neq \alpha_2 \in fv(\Gamma) \cup fv(S),\ \sigma(\alpha_1) = \sigma(\alpha_2) \vee \sigma(\alpha_1) \notin \mathcal{V}) \\ A \vee B\end{array}}{\Gamma \vdash D : S \dashrightarrow \sigma(\Gamma) \vdash D : \sigma(S) \cup \mathrm{Gen}(\sigma(\Gamma), \sigma(S'))}$$

P:CHANNEL

$$\frac{\exists e \in \mathtt{I}(D) \setminus dc(S)}{\Gamma \vdash D : S \dashrightarrow \Gamma \vdash D : S \cup e}$$

Figure 14. The propagation algorithm

Propagation Properties The propagation algorithm being only a type inference *semi*-algorithm, we present here just one theorem, stating how it behaves for typable programs. This theorem is based on the definition of a *propagation error*, which states when the propagation algorithm fails, and of a *terminal* statement, indicating a successful computation.

DEFINITION 5.2.3. *Let* $\Gamma \vdash D : S$ *be a typing statement. We say that a* propagation error *occurs on this statement iff there exist:*

- *A sub-program* $(e_1 : x_1; \ldots; e_n : x_n/R) \subset D$*.*
- *A tuple* $(T_1, \ldots, T_n) \in S(e_1) \times \cdots \times S(e_n)$*.*
- *A valid statement* $fv(\Gamma) \cup fv(S), \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash R : [F]\ S' \mid C$ *such that there is no substitution* σ *with* $C \Rightarrow \sigma$*.*

DEFINITION 5.2.4. *A typing statement* $\Gamma' \vdash D : S'$ *is* terminal *iff no propagation error occurs on it, and no propagation rule can be applied.*

The main property of our propagation algorithm can now be stated:

THEOREM 5.2.1. *Let suppose given a valid typing statement* $\Gamma \vdash D : S_k$. *Then there exist a* terminal *statement* $\Gamma' \vdash D : S'$ *such that* $(\Gamma, \emptyset) \dashrightarrow^* \Gamma' \vdash D : S'$. *Moreover, the statement* $\Gamma' \vdash D : S'$ *holds, and there exist a permutation* σ *such that* $\Gamma = \sigma(\Gamma')$ *and* $\Gamma \vdash S_k \Leftarrow \sigma(S')$.

The proof of the theorem and of the following corollary can be found in the companion report.

COROLLARY 2. *Let suppose given a valid statement* $\Gamma \vdash D : S$. *Then there exist* S' *such that* $\Gamma \vdash_m D : S'$ *holds.*

Case of non-Termination By construction, the propagation algorithm will not finish if there is an infinite number of message types to compute, i.e. when an infinite number of messages with different structures are sent during the program execution. The following program has such a property:

$$!e(x).\overline{e}\langle\{a = x\}\rangle \mid \overline{e}\langle 1\rangle$$

Indeed, at first, the message present on e is '1' typed int, but after one execution of the receiver, the message will be $\{a = 1\}$ typed $\{a : \text{Pre}(\text{int}); \text{Abs}\}$. Thus, the propagation algorithm will compute iteratively types of the form $\{a : \text{Pre}(\ldots\{a : \text{Pre}(\text{int}); \text{Abs}\}\ldots); \text{Abs}\}$ and never finish, because no message operation error will occur, and each application of the first propagation rule will compute a different type. Note that this program has no valid type: informally, to handle the generality of the messages sent on e, one should use a type of the form $e : (\alpha)$, which is not well-defined.

6. Related work

We have mentioned in the introduction previous work dealing with assemblage issues in component-based communication frameworks. Type systems checking architectural constraints or component assemblages have been the subject of various works in the past decade. For instance, the work done on the Wright language [4] supports the verification of behavioral compatibility constraints in a software architecture. Work on Plastik [15] deals mostly with structural constraints, although in a dynamical setting. Work on ArchJava [3] uses ownership types to enforce communication integrity between components. Other work develops behavioral types for component assembly [9], which is close to the notion of session types as developed e.g. in [41]. None of these type systems allow to capture the errors we deal with in this paper, due to incorrect message manipulation operations. The type system we propose in this paper is more related to the ones defined for PICT [27], the π-calculus [22] or the $\lambda\pi_v$-calculus [40], although with provision for extensible record types that these systems do not have.

We know of no type system that is capable of dealing with our notion of message errors, with the complex data flows that are allowed in our calculus. Indeed, type systems such as [8, 13, 27, 35] are too restrictive concerning data flow manipulation, and cannot adequately deal with *routers* and *multiplexers*. On the other hand, type systems which satisfactorily handle data flows [22, 39, 41] do not take in account structured mutable messages.

Type inference for distributed calculi has been studied for the Join-calculus [10], Mobile Ambients-like calculi [23], Dπ [18], which have an inference algorithm, and PICT, which has not. While the reasons for type inference undecidability in PICT are typically higher-order polymorphism and sub-typing, we believe that in our case it is more related to polymorphic recursion [12]. Indeed, undecidability in our case is caused by channels being mapped to a finite set whose cardinality is not constrained, thus allowing a form of polymorphic recursion in loops. Finally, one can consider the *routing process* present in the calculus as a weak form of *type analysis* [38] on rows.

Our component calculus manipulates extensible records, a feature which it shares with the Piccola calculus [1, 21]. In contrast to our calculus, the Piccola calculus, comprises explicit environments (in its latest version [1]), but is untyped. Our work can actually provide a first basis for typing Piccola programs.

7. Conclusion

Extending our initial work reported in [6], this paper has described an approach and a novel type system to deal with for message errors in assemblages built using a component-based communication framework. Our type system combines *rows* and process types, resulting in constructs similar to set types for process types [2]. Our type system deals both with complex data flows, as one can define in the Dream communication framework, and structured, mutable messages, also as found in the Dream framework. Type inference is undecidable, but we have presented an inference semi-algorithm which makes our approach usable in practice. This is all the more true since we have also developed in the companion report [19] a complete inference algorithm that can, given a candidate program, identify the minimal set of required user annotations and compute a minimal type relative to these annotations, if it exists, or terminate with an error otherwise.

We plan to extend this work in several directions. In particular, we have completed an implementation for a variant of our approach that can be used in conjunction with the Fractal ADL toolset [16]. We are currently applying this implementation for checking Dream and Click assemblages. In the near term, we plan to add operations to the calculus that allow manipulating the structure of an assemblage as in e.g. the Kell calculus [34], to extend the type system so as to capture configuration invariants and constraints, and to deal with errors due to reconfiguration operations.

Acknowledgments

We extend our thanks to the anonymous reviewers for their comments. This work has been supported in part by the European 6th Framework Programme, Priority 2, Information Society Technologies, Project IST 34084 Selfman.

References

[1] F. Achermann and O. Nierstrasz. A calculus for reasoning about software composition. *Theor. Comput. Sci.*, 331(2-3), 2005.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[3] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th European Conf. on Object-Oriented Programming*, 2002.

[4] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. In *ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3*, pages 213–249, July 1997.

[5] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4), 1998.

[6] P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.B. Stefani. Dream Types - A Domain Specific Type System for Component-Based Message-Oriented Middleware. In *4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), in association with ESEC/FSE'05/*, 2005.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[8] L. Cardelli. Types for mobile ambients. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1999.

[9] C. Carrez, A. Fantechi, and E. Najm. Behaviour contracts for a sound assembly of components. In *FORTE*, volume 2767 of *Lecture Notes in Computer Science*. Springer, 2003.

[10] S. Conchon and F. Pottier. Join(x): Constraint-based type inference for the join-calculus. In *10th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2001.

[11] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 301–312, New York, NY, USA, 1998. ACM.

[12] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[13] J. Roger Hindley. *Basic simple type theory*. Cambridge University Press, New York, NY, USA, 1997.

[14] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment : Experience with the Aster Prototype. In *4th Int. Conf. on Configurable Distributed Systems*, 1998.

[15] A. Joolia, T. Batista, G. Coulson, and A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *5th IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE Computer Society, 2005.

[16] M. Leclercq, A. E. Ozcan, V. Quema, and J.B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[17] M. Leclercq, V. Quema, and J.B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), 2005.

[18] C. Lhoussaine. Type inference for a distributed π-calculus. *Sci. Comput. Program.*, 50(1-3), 2004.

[19] M. Lienhardt, A. Schmitt, and J.-B. Stefani. A type system for the DREAM framework, 2007. http://sardes.inrialpes.fr/papers/files/rr-dreamtypes.pdf.

[20] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *ACM Symposium on Operating Systems Principles*, 1999.

[21] M. Lumpe, F. Achermann, and O. Nierstrasz. *A Formal Language for Composition*, chapter 4. Cambridge University Press, 2000.

[22] Sergio Maffeis. Sequence types for the pi-calculus. In *ITRS'04*, volume 136 of *ENTCS*, pages 117–132. Elsevier, 2005.

[23] H. Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In *14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*. Springer, 2005.

[24] H. Miranda, A. S. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems (ICDCS 2001)*. IEEE Computer Society, 2001.

[25] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *ACM Symposium on Operating Systems Principles*, 1999.

[26] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[27] B. Pierce and D. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[28] E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

[29] F. Pottier. A 3-part type inference engine. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782, pages 320–335. Springer Verlag, 2000.

[30] F. Pottier. Simplifying subtyping constraints: A theory. *INFCTRL: Information and Computation (formerly Information and Control)*, 170, 2001.

[31] F. Pottier and D. Rémy. The Essence of ML Type Inference. In *B. Pierce (ed), Advanced Topic in Types and Programming Languages*. MIT Press, 2005.

[32] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[33] D. Rémy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[34] A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*. Springer, 2005.

[35] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1):1, 2007.

[36] C. Szyperski. *Component Software, 2nd edition*. Addison-Wesley, 2002.

[37] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building Adaptive Systems Using Ensemble. *Software – Practice and Experience*, 28(9), 1998.

[38] S. Weirich. Higher-order intensional type analysis. In *11th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2002.

[39] N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 334–345, 2000.

[40] N. Yoshida and M. Hennessy. Assigning types to processes. *Inf. Comput.*, 174(2), 2002.

[41] N. Yoshida and V. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4), 2007.