Typing Component-Based Communication Systems

Michaël Lienhardt, Claudio Mezzina, Alan Schmitt, Jean-Bernard Stefani

Typing Component-Based Communication Systems

Michael Lienhardt, Claudio Antares Mezzina,
Alan Schmitt, and Jean-Bernard Stefani

INRIA, France

Abstract. Building complex component-based software systems, for instance communication systems based on the Click, Coyote, Appia, or Dream frameworks, can lead to subtle assemblage errors. We present a novel type system and type inference algorithm that prevent interconnection and message-handling errors when assembling component-based communication systems. These errors are typically not captured by classical type systems of host programming languages such as Java or ML. We have implemented our approach by extending the architecture description language (ADL) toolset used by the Dream framework, and used it to check Dream-based communication systems.

1 Introduction

Building software systems from components has many benefits [33], including easier maintenance and evolution. However, component-based systems are not exempt from subtle assemblage errors that are not captured by the type systems provided with the implementation languages. These errors are hard to catch because they may be purely an artifact of a faulty assemblage, and thus may arise even if individual components and their interconnections are correct. As noted in [24], this is for instance the case with data manipulation errors. These errors may occur when handling protocol data units in a communication stack built from components or micro-protocols with frameworks like Appia [27], Click [18], Coyote [5], Dream [20], or Ensemble [34].

Dealing with assemblage errors in system software and communication systems has already been approached in five main ways. The first one uses theorem proving to check the expected properties of an assemblage on a formal specification of the behavior of individual components and of the assemblage, as in Ensemble [24]. The second approach uses an architecture description language (ADL) to specify component behaviors and assemblage constraints, typically component dependencies, and to automatically verify the assemblage consistency, as in Aster [15], Knit [30], or Plastik [16]. The third approach relies on type systems for interaction contracts, as in the Singularity system [11] or in web service workflows [14]. The fourth approach uses model checking to verify the expected properties of a formally specified assemblage, as in the Vercors system [3]. A fifth approach relies on property-preserving composition, as described in [4], where it is applied to deadlock-free assemblages.

The theorem-proving approach is comprehensive and can address arbitrary properties, but it requires theorem-proving expertise, which is not readily available for systems programmers. The ADL 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 errors. The type-system approach can be made entirely automatic if type inference is decidable, but the type systems devised so far fail to deal with the data handling errors we consider in this paper. The model-checking approach is automatic, but may require considerable expertise in the property language used, again not necessarily available for systems programmers. The property-preserving composition approach also can be made entirely automatic, for instance using model checking techniques, but to this date does not readily apply to the data handling errors we consider.

We thus propose an extension of the ADL approach with a type analysis devised to deal with a class of data manipulation errors that occur in ill-formed communication systems assemblages. More specifically, our approach involves: (i) the definition of a simple process calculus that allows to specify an operational model of a component assemblage (where program execution is abstracted by a reduction relation); (ii) the definition of a type system, that operates on programs abstracted as terms of the process calculus, and that ensures that typable assemblages do not exhibit the targeted class of errors; (iii) an extension of the target ADL to allow architecture descriptions with process annotations characterizing the abstract behavior of selected components; (iv) the addition of a type analyzer in the ADL assembly toolchain to statically verify component assemblages. Technically, the paper makes two main contributions: (i) we define a novel type system, which combines rows [31] with process types [36, 25], to track message flows in component assemblages; (ii) we define a total type inference algorithm for automatically checking annotated component assemblages.

Outline. The paper is organized as follows. Section 2 details the assemblage verification we target. Section 3 presents the calculus and Section 4 the type system that we use to abstract the behavior of communication components and to characterize them. Section 5 discusses type inference and its implementation in actual assemblage tool chains. Section 6 discusses related work and Section 7 concludes the paper.

2 Assemblages in Dream

To explain the assemblage verifications we target in this paper, we use the example of the Dream framework, which we now briefly present. Dream is a component-based framework, written in Java, designed for the construction of communication systems (protocol stacks, communication subsystems of middleware for distributed execution). It is built on top of the Java implementation of the Fractal component model [6].

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 *channels*. A message is a list of *labeled chunks*, which can be any Java objects including messages. Within a component, messages can be freely manipulated. Basic operations, like removing, adding, or accessing chunks are provided. The Dream framework comprises a library of components that encapsulate functions and behaviors commonly found in communication subsystems. These include: *message queues* that are used to store messages, *transformers* that transform a message received on their single input channel and deliver the result to their single output channel, *routers* that forward messages received on their single input channel to one or several output channels, *multiplexers* that forward messages received on their input channels to their single output channel, *aggregators* that aggregate messages received on one or several input channels and deliver the aggregated message on their single output channel, *deaggregators* that are dual to aggregators, and *conduits* that allow messages to be exchanged between different address spaces.

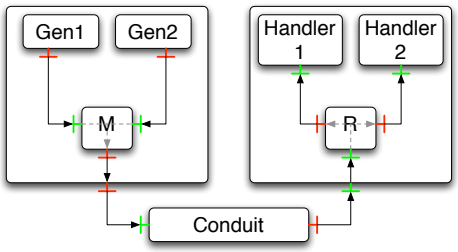

Fig. 1. A DREAM Assemblage

Figure 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 comprises two generator components, `Gen1` and `Gen2`, that emit different messages. These messages are then sent to a multiplexer, then handled by the `Conduit` component 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. Verifying the correctness of the assemblage implies verifying structural constraints to guarantee that input and output channels are properly matched, and ensuring that a component does not receive a message it is not able to handle (typically, a message with missing or unexpected chunks). In our simple example above, this could be the case if the component `Conduit` could not handle messages generated by the two components `Gen1` and `Gen2` (e.g. because of a missing chunk), or if one handler could not process the messages forwarded to it. In the presence of complex assemblages, such an analysis can quickly become difficult.

3 Calculus

Our process calculus aims to capture the abstract behavior of components appearing in communication frameworks. It is at the same level of abstraction than an architecture description language (ADL). Alternatively, it can be understood as a simple ADL. This allows us to apply our approach to different communication frameworks, written in different programming languages.

Syntax The syntax of the calculus is given below. It is parameterized by the set of primitive components (noted p) which can be used in assemblages.

$$
\begin{array}{rlr}
D ::= & & \text{Assemblage} \\
 & p & \text{Primitive} \\
 & \mid\ c[I\,/\,O][D] & \text{Composite} \\
 & \mid\ \overline{e}\langle M\rangle & \text{Message} \\
 & \mid\ D_1 \mid D_2 & \text{Parallel}
\end{array}
\qquad
\begin{array}{rlr}
M ::= & v^{\delta} & \text{Routed value}
\end{array}
$$

$$
\begin{array}{rlr}
\delta ::= & & \text{Tag list} \\
 & \emptyset & \text{Empty tag} \\
 & \mid\ \downarrow r;\delta & \textit{down}\ \text{tag} \\
 & \mid\ \uparrow r;\delta & \textit{up}\ \text{tag}
\end{array}
\qquad
\begin{array}{rlr}
v ::= & & \text{Value} \\
 & c & \text{Base value} \\
 & \{a_1 = v_1; \ldots; a_n = v_n\} & \text{Record}
\end{array}
$$

An assemblage is a parallel composition of components and messages. Components can be primitive or composite. A composite takes the form $c[I\,/\,O][D]$, where c is a name, I is the set of input channels of the composite, O is the set of output channels of the composite, and D is its inner assemblage. The specification of input and output channels I and O in a composite may hide input or output channels of its inner assemblage, by not mentioning them. Messages take the form $\overline{e}\langle M\rangle$, where e is a channel name, and M is a routed value. In the following we write J for a parallel composition of messages. A routed value is a record or a base value decorated with a list of routing tags. We always assume that each tag occur at most once in a list. Intuitively, a list of routing tags δ encodes a particular message flow in a component assemblage. Primitive components can act on these flows, as illustrated by the router and multiplexer primitive components described below. Although each tag is unique in a tag list, component assemblages can contain loops (e.g., through a combination of routers and multiplexers), and record fields can contain records. These two features allow the modeling of complex communication stacks, including ones featuring protocol tunneling, such as IP over IP.

The set of primitive components is a parameter of the calculus, and can be extended as required. It is assumed to contain at least the following primitive components: components `Add`, `Sub`, and `Select` provide classical basic operations on extensible records; components `Router` and `Mult` provide elementary routing and multiplexing capabilities; component `Conn` corresponds to a simple unidirectional connector.

Operational semantics The operational semantics of the calculus is defined classically by a reduction relation between terms that operates modulo a structural equivalence. The structural equivalence is not given here for lack of space (see

[22] for details), but it essentially states that the parallel operator is associative, commutative, and that the order of fields in a record does not matter. The reduction relation is defined as a binary relation on assemblages that satisfies the rules given below. In the rules, a statement of the form "$D_1 \rhd D_2$" can be read "D_1 reduces to D_2".

$$\text{R:Ctx}\ \frac{D \rhd D'}{\mathbb{E}[D] \rhd \mathbb{E}[D']} \qquad \text{R:In}\ \frac{e \in I}{\overline{e}\langle M\rangle \mid c[I\,/\,O][D] \rhd c[I\,/\,O][\overline{e}\langle M\rangle \mid D]}$$

$$\text{R:Out}\ \frac{s \in O}{c[I\,/\,O][\overline{s}\langle M\rangle \mid D] \rhd c[I\,/\,O][D] \mid \overline{s}\langle M\rangle} \qquad \text{R:Prim}\ \frac{\texttt{match}(p, J)}{J \mid p \rhd p \mid \gamma(p, J)}$$

Rule R:Ctx stipulates that reduction is possible inside an evaluation context $\mathbb{E}$ (composite environment or other assemblages in parallel, see [22] for details). Rules R:In and R:Out stipulate how messages flow in and out of composite components. Rule R:Prim is actually a rule schema describing the evolution of primitive components. Informally, it states that if a set of messages J *matches* the input schema of primitive component p (premise $\texttt{match}(p, J)$), then p can consume input messages J and produce output messages described by $\gamma(p, J)$. The relation `match` and the function γ must be defined for all primitive components of interest. For instance, they are defined as follows for `Add`, `Select`, `Mult`, and `Router`. Let $M = \{a_1 = v_1; \ldots; a_n = v_n\}^{\delta_1}$, $R = \{a = v; a_1 = v_1; \ldots; a_n = v_n\}^{\delta_1}$, and a, a_i all distinct. We set:

$$\texttt{match}(\texttt{Add}[e_1\ e_2/s](a), \overline{e_1}\langle M\rangle \mid \overline{e_2}\langle v^{\delta_2}\rangle) \qquad \gamma(\texttt{Add}[e_1\ e_2/s](a), \overline{e_1}\langle M\rangle \mid \overline{e_2}\langle v^{\delta_2}\rangle) = \overline{s}\langle R\rangle$$

$$\texttt{match}(\texttt{Select}[e/s](a), \overline{e}\langle R\rangle) \qquad \gamma(\texttt{Select}[e/s](a), \overline{e}\langle R\rangle) = \overline{s}\langle v^{\delta_1}\rangle$$

$$\texttt{match}(\texttt{Mult}[e_1\ e_2/s](r), \overline{e_1}\langle v^{\delta}\rangle) \qquad \texttt{match}(\texttt{Mult}[e_1\ e_2/s](r), \overline{e_2}\langle v^{\delta}\rangle) \qquad \text{if} \qquad r \notin \delta$$

$$\gamma(\texttt{Mult}[e_1\ e_2/s](r), \overline{e_1}\langle v^{\delta}\rangle) = \overline{s}\langle v^{\uparrow r;\delta}\rangle \qquad \gamma(\texttt{Mult}[e_1\ e_2/s](r), \overline{e_2}\langle v^{\delta}\rangle) = \overline{s}\langle v^{\downarrow r;\delta}\rangle$$

$$\begin{gathered}\texttt{match}(\texttt{Router}[e/s_1\ s_2](r), \overline{e}\langle v^{\delta}\rangle) \qquad \text{if } r \in \delta \\ \gamma(\texttt{Router}[e/s_1\ s_2](r), \overline{e}\langle v^{\delta_1;\uparrow r;\delta_2}\rangle) = \overline{s_1}\langle v^{\delta_1;\delta_2}\rangle \\ \gamma(\texttt{Router}[e/s_1\ s_2](r), \overline{e}\langle v^{\delta_1;\downarrow r;\delta_2}\rangle) = \overline{s_2}\langle v^{\delta_1;\delta_2}\rangle\end{gathered}$$

`Add` and `Select` provide usual record manipulation. `Mult` adds a tag to a routed value to signal the input channel on which it received it. `Router` checks the tags of the received routed values to send them on the appropriate channel.

Errors We say that an assemblage D *cannot process* a message $\overline{e}\langle M\rangle$ if a primitive component p in D may accept a message on e but cannot process the message

$\overline{e}\langle M\rangle$: there are some N and J such that $\texttt{match}(p, \overline{e}\langle N\rangle \mid J)$ but for every J' we don't have $\texttt{match}(p, \overline{e}\langle M\rangle \mid J')$. We then define an assemblage D to be *in error* if $D = \mathbb{E}[\overline{e}\langle M\rangle \mid D']$ and $\overline{e}\langle M\rangle$ cannot be processed by D'. Intuitively, an assemblage is correct if no message manipulation error may occur, i.e., every primitive component that may accept a message can process it.

4 Types

4.1 Type System

Syntax Our type system is based on two main ideas: (i) the type of values exchanged on channels are *routed types*: rows (extensible record) or base types, decorated with routing information; (ii) the type of an assemblage is an *assemblage type*, presented as a function from its input channel types to its output channels types. The syntax of types is defined below.

$E ::=$		Value type	$T ::=$		Routed type
	η	Variable		$\xi[E]$	Value flow
	$\mid \{W\}$	Row		$\mid r(T_1, T_2)$	Tagged pair
	$\mid \tau$	Base type			
			$S ::=$		Channel type
$W ::=$		Row definition		$\emptyset$	Empty declaration
	ρ	Row variable		$\mid e : (T)$	Channel declaration
	$\mid a : \mathrm{Pre}(E); W$	Used Field		$\mid S \cup S$	Union
	$\mid a : \mathrm{Abs}; W$	Unused Field			
	$\mid \mathrm{Abs}$	Empty Row			

The type of an assemblage, written F in the following, takes the form of a type scheme $\forall \alpha_1 \ldots \alpha_n . S_I \to S_O$ where α_i are type variables (standing for arbitrary types), S_I collects the types of input channels in the assemblage, and S_O collects the types of output channels in the assemblage. We write $dc(S)$ for the channel names that appear in S. A channel type takes the form $e : (T)$, where e is a channel name, and T is a routed type. A routed type is either a value flow $\xi[E]$, where the value type E is carried by the data flow ξ, or a tagged pair of the form $r(T_1, T_2)$, where r is a tag, and T_1, T_2 are routed types. Rows are defined classically [31] with presence and absence information: $a : \mathrm{Pre}(E)$ stands for a field named a that is present in a record, with type E; $a : \mathrm{Abs}$ indicates that field a is not present. Base types, i.e., types associated with base values, are a parameter of the type system (base types typically include integers, strings, or concrete data types).

Informally, a routed type is a binary tree where each leaf corresponds to a value type carried by a data flow, and the branch leading to it defines the routing annotation carried by the value (a given routing tag appears at most once on each branch). For instance, the type $r_1(\xi_1[\mathrm{int}], r_2(\xi_2[\mathrm{string}], \xi_2[\eta]))$ consists of three branches corresponding to three different values. The second branch $r_1(\_, r_2(\xi_2[\mathrm{string}], \_))$ corresponds to a flow accepting only strings tagged with at

least the tags $\downarrow r_1$ and $\uparrow r_2$. This tree structure uses explicit references to data flows as they enable *type duplication*, which is a requirement to properly deal with routing and multiplexing. Type duplication allows two multiplexers in a row to type check correctly and is the main innovation of this type system (see the discussion in Section 4.2).

Typing Types for primitive components are given by a function Υ that maps primitive components to assemblage types. Just as the set of primitive components is a parameter of our calculus, function Υ is a parameter of our type system and needs to be defined for every primitive component to be typed. To ensure that these assemblage types correspond to the operational semantics of the primitive components, the function Υ must obey two constraints: (i) for each primitive component p, the input channel type of $\Upsilon(p)$ should only allow valid patterns; (ii) the output type of the parallel composition of a primitive component p with one of its valid input pattern J must contain the type of $\gamma(p, J)$. Formally, for all primitive component p and all J with $\texttt{match}(p, J)$, there exists an assemblage type $S_1 \to S_2$ such that $p \mid J : S_1 \to S_2$ holds, and there exists S_2' with $S_2' \subset S_2$ such that $p \mid \gamma(p, J) : S_1 \to S_2'$ holds. These constraints ensure that the type of a primitive component is consistent with its behavior (defined by relation `match` and function γ). For instance, the types associated with the primitive components introduced before, and of a simple connector $\texttt{Conn}[e/s]$ (that forward any value received on its input channel e to its output channel s), can be defined as follows:

$\Upsilon(\texttt{Add}[e_1 e_2/s](a)) = \forall \alpha, \rho, \xi.\ e_1 : (\xi_1[\{a : \text{Abs}; \rho\}]) \cup e_2 : (\xi_2[\alpha]) \to s : (\xi_1[\{a : \text{Pre}(\alpha); \rho\}])$
$\Upsilon(\texttt{Select}[e/s](a)) = \forall \alpha, \rho, \xi.\ e : (\xi[\{a : \text{Pre}(\alpha); \rho\}]) \to s : (\xi[\alpha])$
$\Upsilon(\texttt{Router}[e/s_1\, s_2](r)) = \forall \alpha, \beta, \xi, \xi'.\ e : (r(\xi[\alpha], \xi'[\beta])) \to s_1 : (\xi[\alpha]) \cup s_2 : (\xi'[\beta])$
$\Upsilon(\texttt{Mult}[e_1\, e_2/s](r)) = \forall \alpha, \beta, \xi, \xi'.\ e_1 : (\xi[\alpha]) \cup e_2 : (\xi'[\beta]) \to s : (r(\xi[\alpha], \xi'[\beta]))$
$\Upsilon(\texttt{Conn}[e/s]) = \forall \alpha, \xi.\ e : (\xi[\alpha]) \to s : (\xi[\alpha])$

The type system is equipped with a (classical) subtyping relation $\leq$, which we do not detail fully here, for lack of space. For instance, the subtyping rules for assemblage types T:Func and T:Gen, and tagged pairs T:TagPair, are given below (note the contravariance in T:Func, which is as expected):

$$\text{T:Func}\ \frac{S_1 \leq S_1' \qquad S_2 \leq S_2'}{S_1' \to S_2 \leq S_1 \to S_2'} \qquad \text{T:Gen}\ \frac{F \leq F'}{\forall \alpha.F \leq \forall \alpha.F'} \qquad \text{T:TagPair}\ \frac{T_1 \leq T_1' \qquad T_2 \leq T_2'}{r(T_1, T_2) \leq r(T_1', T_2')}$$

The typing rules in our type system comprise rules for assemblages and rules for routed values. Typing judgements take the form $D : F$ for assemblages, $v : E$ for simple values, and $\mathcal{R} \vdash R : T$ for routed values. The environment $\mathcal{R}$ is a set of routing tags. The typing rules make use of the $\precsim$ binary relation between channel types, which is defined as follows: given two channel types $S \triangleq \bigcup_{i \in I} e_i : (T_i)$ and $S' \triangleq \bigcup_{j \in J} e_j' : (T_j')$, we note $S \precsim S'$ iff for all $i \in I, j \in J$, $e_i = e_j'$ implies $T_i \leq T_j'$.

Typing rules for assemblages are given below:

$$\text{T:Prim}\ \frac{\Upsilon(p) = F}{p : F} \qquad \text{T:Subst}\ \frac{D : F}{D : \sigma(F)} \qquad \text{T:Inst}\ \frac{D : \forall\alpha.F}{D : F} \qquad \text{T:Gen}\ \frac{D : F}{D : \forall\alpha.F} \qquad \text{T:Channel}\ \frac{\emptyset \vdash M : T}{\overline{e}\langle M\rangle : \emptyset \to e : (T)}$$

$$\text{T:Sub}\ \frac{D : F \qquad F \leq F'}{D : F'} \qquad \text{T:Par}\ \frac{D : S_1 \to S_2 \qquad D' : S_1' \to S_2' \qquad S_2 \precsim S_1' \qquad S_2' \precsim S_1 \qquad dc(S_1) \cap dc(S_1') = \emptyset}{D \mid D' : (S_1 \cup S_1') \to (S_2 \cup S_2')}$$

$$\text{T:Box}\ \frac{D : S_1 \to S_2 \qquad S_1' \precsim S_1 \qquad S_2 \precsim S_2' \qquad S_2' \precsim S_1' \qquad dc(S_1') = I \wedge dc(S_2') = O}{c[I\,/\,O][D] : S_1' \to S_2'}$$

Rule T:Prim states that the type of a primitive component is given by function Υ. Rules T:Subst, T:Inst, and T:Gen are classical rules for substitution, instantiation, and generalization, respectively. Since type duplication is integrated into substitutions, because of the different forms of type variables and their associated constraints (e.g., unique occurrence of tags in routing annotations), our notion of substitution σ in rule T:Subst is slightly more complex than usual. It mostly behaves as expected, replacing variables with terms (see discussion in Section 4.2; formal details can be found in [22]).

The parallel composition D_1 of two assemblages D and D' yields a function having the *capacity* of both assemblages, i.e. , that accepts as input any message either D or D' accepts, and that can generate any message either D or D' can generate. Rule T:Par has three side conditions: the first two ($S_2 \precsim S_1'$ and $S_2' \precsim S_1$) ensure that all values (S_2 and S_2') sent on input channels for $D \mid D'$ are indeed valid inputs for this program; the third one ($dc(S_1) \cap dc(S_1') = \emptyset$) states that D and D' must have distinct input channels to avoid the possibility of *implicit routing*, i.e. , of distinct components listening on the same channel, thus doing a routing operation without an explicit router to support it. Rule T:Box specifies the constraints that apply to obtain the type $S_1' \to S_2'$ of a composite. The sets S_1' and S_2' must give a type to every channel mentioned in I and O. If a channel is mentioned in both, then the output type must be a subtype of the input type ($S_2' \precsim S_1'$) as this corresponds to a loop We also impose that the valid inputs of the component must be valid ones for the component's inner process (stated by the constraint $S_1' \precsim S_1$), and that all outputs of this process must be valid output of the component (stated by the constraint $S_2 \precsim S_2'$).

Typing rules for routed values are given below (we have left out rules and conditions that apply to base values and base types):

$$\text{T:RECORD}\quad \frac{\forall 1 \le i \le n,\, v_i : E_i \qquad \forall 1 \le i \neq j \le n,\, a_i \neq a_j}{\{a_1 = v_1; \ldots; a_n = v_n\} : \{a_1 : \mathrm{Pre}(E_1); \ldots; a_n : \mathrm{Pre}(E_n); \mathrm{Abs}\}} \qquad \text{T:EMPTY}\quad \frac{v : E}{\mathcal{R} \vdash v^{\emptyset} : \xi[E]}$$

$$\text{T:UP}\quad \frac{\mathcal{R} \uplus \{r\} \vdash v^{\delta} : T \qquad \mathcal{R} \uplus \{r\} \vdash T_k}{\mathcal{R} \vdash v^{\uparrow r;\delta} : r(T, T_k)} \qquad \text{T:DOWN}\quad \frac{\mathcal{R} \uplus \{r\} \vdash v^{\delta} : T \qquad \mathcal{R} \uplus \{r\} \vdash T_k}{\mathcal{R} \vdash v^{\downarrow r;\delta} : r(T_k, T)}$$

Rule T:RECORD is the standard typing rule for extensible record, using rows. The three typing rules T:EMPTY, T:UP and T:DOWN, construct a routed type by induction on the cardinality of the routing annotation. Rule T:EMPTY is used when the routing annotation is empty: the routing type is in such case just a leaf representing the value's type. Rules T:UP and T:DOWN define how we construct the routing type tree when one or more elements are present in the routing annotation. We write $\mathcal{R} \uplus \{r\}$ for the disjoint union of the tow sets. Generic flows that are built in a routing type derivation will then be instantiated during the exploration of the rest of the program with the typing rule T:INST. The use of routing tags environments $\mathcal{R}$ in these three rules ensures the validity of the constructed routed type.

Example assemblage Assume that the generators, handlers, multiplexer, router and conduit components in Figure 1 are primitive components, and their types are as given in the following table. We can type the assemblages `SiteA` and `SiteB` as indicated in the last two lines of the same table.

Component	Types
`Gen1`	$\forall \xi.\emptyset \to s_1 : (\xi[E_1])$
`Gen2`	$\forall \xi.\emptyset \to s_2 : (\xi[E_2])$
`Handler1`	$\forall \xi.e_1 : (\xi[E_3]) \to \emptyset$
`Handler2`	$\forall \xi.e_2 : (\xi[E_4]) \to \emptyset$
`M`	same type as $\texttt{Mult}[s_1\, s_2/t_A](r)$
`R`	same type as $\texttt{Router}[t_B/e_1\, e_2](r)$
`Conduit`	same type as $\texttt{Conn}[t_A/t_B]$
`SiteA`	$\forall \xi.\emptyset \to t_A : (r(\xi[E_1], \xi'[E_2]))$
`SiteB`	$\forall \xi.t_B : (r(\xi[E_3], \xi'[E_4])) \to \emptyset$

If we assume further that E_3 can be transformed using sub-typing and substitution into E_1, and similarly for E_4 into E_2, then we can type the (closed) assemblage

$$c[\emptyset\,/\,\emptyset][\texttt{SiteA} \mid \texttt{Conduit} \mid \texttt{SiteB}]$$

with the type: $\emptyset \to \emptyset$.

Properties of the type system The type system is sound with respect to reduction and guarantees correct execution, as shown by the subject reduction and correction theorems, and type inference is decidable (see proofs in [22]):

Theorem 1 (Subject Reduction). *Let D and D' be two assemblages such that $D \triangleright D'$, and F an assemblage type such that $D : F$ holds. Then there exists F' such that $D' : F'$.*

Theorem 2 (Correction). *Let D be an assemblage and F a process type such that $D : F$ holds. Then D has no error.*

Theorem 3 (Inference). *Type inference is decidable.*

4.2 Discussion

Type duplication. In our presentation of the type system, we have, for lack of space, glossed over several details (which can be found in [22]). In particular, our notion of substitution is more complex than the usual one because of type duplication. Let us explain this by way of an example. One of the objectives of this type system was to allow flexible data flows in programs, using a routing tree structure to type our channels. Let us consider a program where a component `Rem` that remove a a field follows a multiplexer. The output type of the multiplexer is of the form $r(\xi_1[\eta_1], \xi_2[\eta_2])$, whereas the input type of `Rem` is of the form $\xi_3[\{a : \mathrm{Pre}(\eta_3); \rho\}]$. The difficulty here is that we need to be able to unify these two types to get a valid type system. With our definition of substitution, this unification is made in two steps. We first *duplicate* the type $\xi_3[\{a : \mathrm{Pre}(\eta_3); \rho\}]$ into

$$T \triangleq r(\xi_4[\{a : \mathrm{Pre}(\eta_3); \rho\}], \xi_5[\{a : \mathrm{Pre}(\eta_3); \rho\}])$$

One can remark that the two branches of the resulting routing tree have the same row and type variables. But because they are declared in different flows (ξ_4 and ξ_5), they can be instantiated with different terms. We then have two tree structures with the same form that we can simply unify into T.

Duplication allows to instantiate a leaf in a routing tree into a whole subtree, while keeping the constraint of the leaf (here, the constraint being that the message must have the field 'a' defined) and allowing the variables on the fresh leaves to be instantiated independently. One can see duplication as a way to enable polymorphism without using type schemes.

Routing on tags. One may notice that routing in our calculus is based on routing tags, and not, as could be envisaged, on message values or on (the presence of) fields in record values. Likewise, the type system could depend only on rows for message types. In fact, an earlier version of our calculus and type system did exactly that, and is described in [23]. Both calculus and type system in this earlier version are more expressive than the ones presented here. For instance, the type system in [23] allows types associated with a single channel to be union types, in contrast to the type system in this paper. Unfortunately, for reasons

explained in [23], type inference in our earlier type system is undecidable. Our calculus and type system in the present paper thus trade expressivity in favor of the decidability of inference, which is ultimately due to the fact that routing types are finite trees.

Limitations. Our type system has a few limitations. We already pointed out that there can only be a single type for channel and a set of tags (union types are not supported). Also, since a routing type is a binary tree, one has to encode router and multiplexer types with more than two output or input channels by a combination of binary routers and multiplexers. Another consequence is the complexity of encoding routers that route on fields into our calculus, as is the case in the Dream framework. Typically, we encode the presence of a field a in a message with a pair of tags $\uparrow a$ (when the field a is present) and $\downarrow a$ (when a is absent from the message). This simple encoding is difficult to apply in complex assemblages involving loops with multiple routers and multiplexers. An encoding can be found in most cases, but can be tricky to define and manipulate. However, based on our experience with the Dream and Click frameworks (see below), these limitations are not show-stoppers, and we have not in practice encountered the difficult cases mentioned above.

5 Type Inference and its Implementation

A key property of our type system (in contrast to our previous work [23]) is that type inference is decidable. We have devised and proved correct a constraint-based algorithm, along the lines of [28, 9]. We do not have the space to present the type inference algorithm: its definition and proof can be found in [22]. The algorithm comprises a constraint generator that computes from a given program a set of constraints a type must satisfy to type the input program, and a constraint solver that decides whether the generated constraint set has a solution (the program is typable) or not (the program is not typable). Technically, our type inference is based on the one defined in [9], extended to deal with routing types, channel types, and type duplication.

We have implemented the type inference algorithm in OCaml, and used it to extend the assemblage tool chains used by the Dream and Click frameworks. In the case of Dream, we have extended the Fractal ADL toolchain described in [19]. Figure 2 provides an overview of this toolchain. It is organized as a component-based framework, that comprises essentially a front-end, realized by the `Loader` component in Figure 2, and a back-end, that comprises the `ASTProcessingOrganizer` and the `Scheduler` components in Figure 2. The back-end is responsible for the generation and execution of tasks such as code generation, code installation, code deployment, etc. The `Loader` component reads a set of input files and produces an Abstract Syntax Tree (AST). This tree provides a unified representation of the system architecture that can be described through a combination of description languages, such as ADL, IDL, or DSL. The `Loader` is organized essentially as a pipeline comprising parsers

for the various possible input languages, and semantic analyzers. We have integrated our type analyzer as a specific semantic analyzer component in this pipeline. We have also devised an extension to the XML-based Fractal ADL to take into account our type annotations for primitive components, and added its associated parser component in the `Loader` pipeline.

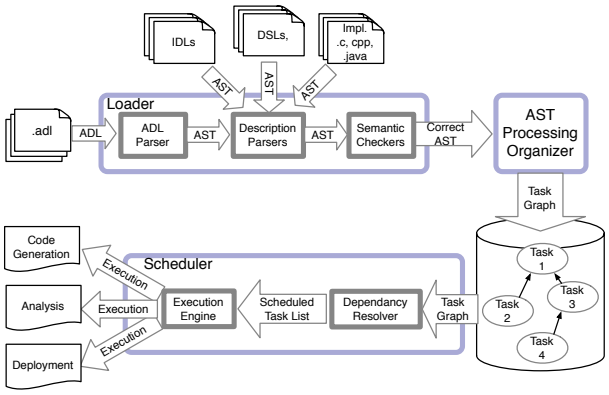

Fig. 2. Fractal ADL toolchain

In the case of Click, a C++ software framework dedicated to the component-based construction of configurable routers [18], assemblages are specified by configuration files written in a simple scripting language [17]. We found it simpler to just document type annotations for Click in a separate, additional configuration file. This way, our type analyzer remains an entirely separate and external analysis tool for Click, and its use does not require any change to the Click toolset.

We also conducted several experiments to check the correctness of non-trivial assemblages built using both frameworks. We have no space to report fully on these experiments but they demonstrate that our approach is practical, requiring minimal extensions to existing assemblage toolsets, and that it can indeed be applied to different component-based frameworks, implemented in different programming languages. The following table provides an indication of the time taken to check (correct) Dream and Click assemblages. The Dream assemblage originates from the Cosmos project, which develops protocols for roaming mobile devices. The Click assemblages are examples taken from the Click website. The performance of our type analyzer appears quite reasonable, bearing in mind that the complexity of type inference in our system is non-polynomial.

Assemblage	Components	Primitive	Channels	Time (sec)
COSMOS (Dream)	439	340	662	180.428
dnsproxy (Click)	9	8	7	0.025
fromhost-tunnet (Click)	24	22	24	0.166
mazu-nat (Click)	60	56	54	4.489

6 Related Work

Type systems checking architectural constraints or component assemblages have been the subject of several works in the past decade. For instance, the work done on the Wright language [2] supports the verification of behavioral compatibility constraints in a software architecture. Work on Plastik [16] deals mostly with structural constraints, although in a dynamical setting. Work on ArchJava [1] uses ownership types to enforce communication integrity between components. Another work develops behavioral types for component assembly [8], which is close to the notion of session types as developed in [38]. None of these type systems capture the errors we deal with in this paper, namely incorrect message manipulation operations. The type system we propose in this paper is more related to the ones defined for PICT [29], the π-calculus [25] or the $\lambda\pi_v$-calculus [37], 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 along with the complex data flows that are allowed in our calculus. Indeed, type systems such as [7, 13, 29, 32] are too restrictive concerning data flow manipulation, and cannot adequately deal with *routers* and *multiplexers*. On the other hand, type systems which provide some means to handle data flows by way of session types and process types [8, 25, 36, 38] do not take in account structured mutable messages.

Type inference for distributed calculi has been studied in the setting of the Join-calculus [10], Mobile Ambients-like calculi [26], Dπ [21], which have an inference algorithm, and PICT, which has not. In our earlier work [23], type inference was undecidable. Undecidability was caused by channels being mapped to a finite set whose cardinality is not constrained, thus allowing a form of polymorphic recursion in loops [12]. In the present work, because of the use of tags, we only allow a kind of *finite* polymorphism in loops, thus obtaining decidable type inference. Finally, one can consider the *routing process* present in the calculus as a weak form of *type analysis* [35] on rows.

7 Conclusion

We have presented in this paper an approach and a novel type system to deal with data handling errors that may occur in communication systems built with component-based communication frameworks. Our approach, which can be characterized as a domain-specific type analysis, extends previous approaches based on architecture descriptions analysis, to deal with both structural and behavioral errors. It complements structural verifications that are the traditional remit of ADL-based approaches, and can as well be an interesting complement to behavior verification tools based on model-checking. We have implemented a type analyzer tool that comprises a total type inference algorithm for component assemblages, and applied it to the checking of several configurations built with two different communication frameworks. These experiments demonstrate, in our view, that our approach is indeed promising and practical.

We plan to extend this work in several directions. We are currently trying to generalize the notion of tagged types in order to apply to concurrent functional languages, and to extend our approach to deal with reconfiguration errors in dynamically evolving assemblages.

References

1. J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th European Conf. on Object-Oriented Programming*, 2002.
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. In *ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3*, 1997.
3. T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model-checking Distributed Components: The Vercors Platform. *Electr. Notes Theor. Comput. Sci.*, 182, 2007.
4. S. Bensalem, M. Bozga, J. Sifakis, and T.H. Nguyen. Compositional Verification for Component-Based Systems and Application. In *ATVA 2008*, volume 5311 of *LNCS*. Springer, 2008.
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. 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.
7. L. Cardelli. Types for mobile ambients. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1999.
8. C. Carrez, A. Fantechi, and E. Najm. Behaviour contracts for a sound assembly of components. In *FORTE*, volume 2767 of *LNCS*. Springer, 2003.
9. François Pottier. A constraint-based presentation and generalization of rows. *Symposium on Logic in Computer Science (LICS)*, 2003.
10. S. Conchon and F. Pottier. JOIN(X): Constraint-Based Type Inference for the Join-Calculus. In *10th European Symp. on Programming (ESOP)*, 2001.
11. M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *1st EuroSys Conference*. ACM, 2006.
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. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *35th ACM Symposium on Principles of Programming Languages (POPL 2008)*. ACM, 2008.
15. 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.
16. A. Joolia, T. Batista, G. Coulson, and A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *WICSA'05)*. IEEE Computer Society, 2005.
17. E. Kohler, R. Morris, and B. Chen. Programming language optimizations for modular router configurations. In *ASPLOS*, 2002.

18. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3), 2000.
19. M. Leclercq, A. E. Ozcan, V. Quema, and J.B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *29th Int. Conf. on Soft. Eng. (ICSE)*. IEEE Computer Society, 2007.
20. 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.
21. C. Lhoussaine. Type inference for a distributed π-calculus. *Sci. Comput. Program.*, 50(1-3), 2004.
22. M. Lienhardt, C.A. Mezzina, A. Schmitt, and J.B. Stefani. Typing communicating component assemblages v2. `http://sardes.inrialpes.fr/papers/dtv2.pdf`, 2008.
23. M. Lienhardt, A. Schmitt, and J.B. Stefani. Typing communicating component assemblages. In *GPCE 2008*. ACM, 2008.
24. 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.
25. Sergio Maffeis. Sequence types for the pi-calculus. In *ITRS'04*, volume 136 of *ENTCS*, pages 117–132. Elsevier, 2005.
26. 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 *LNCS*. Springer, 2005.
27. H. Miranda, A. S. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS 2001*. IEEE Computer Society, 2001.
28. Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.
29. 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.
30. A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. In *OSDI 2000*, 2000.
31. 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.
32. V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1):1, 2007.
33. C. Szyperski. *Component Software, 2nd edition*. Addison-Wesley, 2002.
34. R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building Adaptive Systems Using Ensemble. *Software – Practice and Experience*, 28(9), 1998.
35. S. Weirich. Higher-order intensional type analysis. In *11th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2002.
36. N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.
37. N. Yoshida and M. Hennessy. Assigning types to processes. *Inf. Comput.*, 174(2), 2002.
38. 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.