Dream Types

A Domain Specific Type System for
Component-Based Message-Oriented Middleware

Philippe Bidinger, Matthieu Leclercq, Vivien Quéma, Alan Schmitt, Jean-Bernard Stefani

Projet Sardes, INRIA Rhône-Alpes

ABSTRACT

We present a type system for the Dream component-based message-oriented middleware. This type system aims at preventing the erroneous use of messages, such as the access of missing content. To this end, we adapt to our setting a type system developed for extensible records.

1. INTRODUCTION

Component-based frameworks have emerged in the past two decades. They are commonly used to build various software systems, including Web applications (EJB [1], CCM [14]), middleware (dynamicTAO [11], OpenORB [4]), or even operating systems (OSKit [10], THINK [9]).

A typical example of such frameworks is Dream [12]. Dream allows the construction of message-oriented middleware and builds upon the Fractal component model [5] and its Java implementation. It provides a library of components that that can be assembled using the Fractal architecture description language (ADL) and that can be used to implement various communication paradigms, such as message queues, event/reaction, publish/subscribe, etc.

A system built out of Dream components typically comprises several components which may exchange *messages*, which may modify them (*e.g.*, setting a time stamp), and which may behave differently according to their contents (*e.g.*, routing a message). In the current Java implementation of the Dream framework, every message has type `Message`, independently of its contents. As a consequence, certain assemblages of Dream components type-check and compile correctly in Java but lead to run-time failures, typically when a component processes a message that does not have the proper expected structure.

Catching such configurations errors early on, when writing the architecture description of a Dream assemblage, would be of tremendous benefits to programmers using the Dream framework. In other words, what would be required would be a type-safe ADL that would allow the typing of component structures and reject ill-typed component configurations.

As a first step towards this goal, we propose in this paper a type system for Dream components, concentrating on message types that accurately describe the internal structure of a message. To this end, we adapt existing work on type systems for *extensible records* [16, 17] and describe how components and component assemblages may be typed. The resulting type system captures a number of errors that can be made when writing ADL descriptions of Dream configurations.

The paper is structured as follows: Section 2 describes the Dream framework, and typical configuration errors the type system is intended to capture. Section 3 introduces types for messages and for components manipulating messages. Section 4 describes related work, and Section 5 concludes the paper.

2. THE DREAM FRAMEWORK

2.1 The Fractal component model

Dream is based on the Fractal component model [5], a component model for Java. Fractal distinguishes between two kinds of components: *primitive* components and *composite* components. The latter provide a means to deal with a group of components as a whole.

A component has one or more ports that correspond to access points supporting a finite set of methods. Ports can be of two kinds: server ports, which correspond to access points accepting incoming method calls, and client ports, which correspond to access points supporting outgoing method calls. The signatures of both kinds of ports is described by a standard Java interface declaration, with an additional role indication (server or client).

A component is made of two parts: the *content part* is either a standard Java class (in the case of a primitive component), or a set of sub-components (in the case of a composite component); the *controller part* comprises interceptors and controllers. Examples of controllers are the *binding controller* that allows binding and unbinding the component's client ports to server ports of other components, and the *lifecycle controller* that allows starting/stopping components.

Figure 1 illustrates the different constructs in a typical Fractal component architecture. Thick gray boxes denote the controller part of a component, while the interior of these boxes correspond to the content part of a component. Arrows correspond to bindings, and tee-like structures protruding from gray boxes are ports.

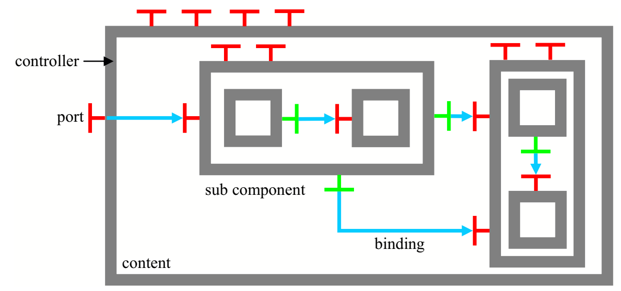
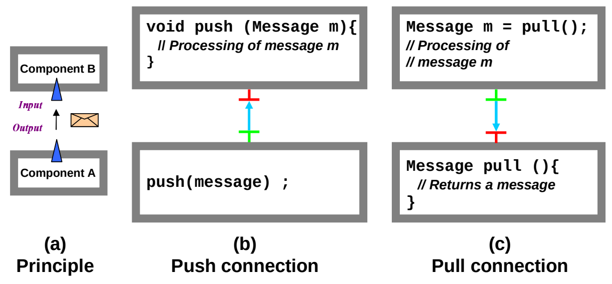


Figure 1: Architecture of a Fractal component

2.2 The Dream framework

Dream components are standard Fractal components with one characteristic feature: the presence of input/output interfaces that allow Dream components to exchange *messages*. Messages are Java objects that encapsulate named *chunks*. Each chunk implements an interface that defines its type. As an example, messages that need to be causally ordered have a chunk that implements the `Causal` interface. This interface defines methods to set and get a matrix clock.

Messages are always sent from outputs to inputs (Figure 2 (a)). There are two kinds of output and input interfaces, corresponding to the two kinds of connections: *push* and *pull*. The push connection corresponds to message exchanges initiated by the output port (Figure 2 (b)). The pull interaction corresponds to message exchanges initiated by the input port (Figure 2 (c)).



```
void push (Message m){
  // Processing of message m
}
```

```
push(message) ;
```

```
Message m = pull();
// Processing of
// message m
```

```
Message pull (){
  // Returns a message
}
```

Figure 2: Input/output interfaces connection

Dream provides a library of components encapsulating functions and behaviors commonly found in message-oriented middlewares. These components can be assembled to implement various asynchronous communication paradigms: message passing, publish/subscribe, event/reaction, etc. Here are a few examples of Dream components:

- *message queues* are used to store messages. Several kinds exist, differing by the way messages are sorted: FIFO, LIFO, causal order, etc.
- *transformers* have one input to receive messages and one output to deliver transformed messages. Typical transformers include stampers.
- *routers* have one input and several outputs (also called "routes"), and route messages received on their input to one or several routes.
- *multiplexers* have several inputs and one output; for every message received on an input a multiplexer adds a chunk that identifies the input on which the message arrived; the multiplexer then forwards the message to the output.
- *duplicators* have one input and several outputs, and copy messages they receive on their input to all their outputs.
- *channels* allow message exchanges between different address spaces. Channels are distributed composite components that encapsulate, at least, two components: a *ChannelOut*—whose role is to send messages to another address space—, and a *ChannelIn*—which can receive messages sent by the ChannelOut.

2.3 Configuration errors

The main data structures manipulated by Dream components are *messages*. A message is a finite set of named *chunk*. A chunk can be any Java object. Basic operations over messages allow to read, remove, add, or update a chunk of a given name. They can potentially lead to three kinds of run-time errors.

- A chunk is absent when it should be present (*e.g.*, for a read, remove, or update).
- A chunk is present when it should be absent (*e.g.*, for an add).
- A chunk does not have the expected type (*e.g.*, for a read).

Experience with the dream framework has shown that many such errors are consequences of an erroneous architecture definition of the system. For instance, in figure 3, the architecture definition is obviously incorrect: component `readTS` expects messages with a `TS` chunk, whereas component `addTS` expects messages without `TS` chunk. Since both components receive exactly the same messages (duplicated by the `duplicate` component), one of them will fail.

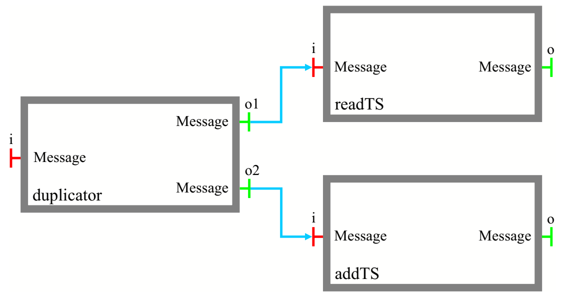


Figure 3: Example

One can tell that the architecture definition of 3 is incorrect because the behavior of the components is clear from their name. However, the typing annotations are clearly insufficient to allow the previous analysis.

In the current component model of Dream, connections between components are constrained by the host language (*e.g.* Java) type system. Ports are associated to Java interface types, and two ports can be connected if and only if their corresponding Java types coincide. This scheme suffers from limitations of the Java type system, in particular, the absence of polymorphism and rich record types.

We propose to define a polymorphic type system for the composition of components in order to overcome those limitations. It allows the specification of the more common behaviors of Dream components, seen as messages transformers. It provides the guarantees that, if components conform individually to their type, the composed system will not fail with any of the run-time errors identified above.

3. DREAM TYPES

3.1 Presentation

A record is a finite set of associations, called *fields*, between labels and values. Many languages, such as Ocaml, use records as primitive values. In [16, 17] Rémy describes an extension of ML where all common operations on records are supported. In particular, the addition or removal of fields and the concatenation of records. He then defines a static type system that guarantee that the resulting programs will not produce run-time errors, such accessing a missing field.

Dream messages can be seen as records, where each chunk correspond to a field of the record, and Dream components can be seen as polymorphic functions. Polymorphism is important for at least two reasons. First, the same component can be used in different contexts with different types. Second, polymorphism allows to relate the types of the client and server interfaces, and thus allows to specify more precisely the behavior of a component. We can almost directly use the results of [16, 17] in order to type Dream components. Note however that we work on a different level of abstraction: we give types to components and check that the way we connect them is coherent. In particular, we do not type-check the code of the components.

In the following, we first give the main ideas behind messages types and component types, and present the main formal results in the next subsection.

We type messages as extensible records [16]. Informally, The type of a message consists of a list of pairwise distinct labels together with the type of the corresponding value, or a special tag if the message does not contain a given label. Moreover, a final information specifies the content of the (infinitely many) remaining labels. In addition, we use a convenient type constructor `ser`: if τ is an arbitrary type, $\mathtt{ser}(\tau)$ is the type of values of type τ in a serialized form.

Figure 4 defines several examples of message types.

$$
\begin{aligned}
\mu_1 &= \{a:\mathtt{pre}(\mathtt{A}); b:\mathtt{pre}(\mathtt{B}); \mathtt{abs}\}\\
\mu_2 &= \{a:\mathtt{pre}(\mathtt{A}); b:\mathtt{pre}(\mathtt{B}); c:\mathtt{abs}; \mathtt{abs}\}\\
\mu_3 &= \{a:\mathtt{pre}(X); \mathtt{abs}\}\\
\mu_4 &= \{a:Y; \mathtt{abs}\}\\
\mu_5 &= \{a:\mathtt{pre}(\mathtt{A}); Z\}\\
\mu_6 &= \{a:\mathtt{pre}(\mathtt{A}); b:Z'; Z''\}\\
\mu_7 &= \{a:\mathtt{pre}(\mathtt{A}); a:\mathtt{pre}(\mathtt{B}); \mathtt{abs}\}\\
\mu_8 &= \{a:X; b:\mathtt{abs}; X\}
\end{aligned}
$$

Figure 4: Examples of message types

A message m of type μ_1 contains exactly two labels a and b, associated to values of type `A` and `B` respectively (the importance of the `pre` constructor will be made clear later). It does not contain any other label, as specified by the `abs` tag. We can note that m can equivalently be seen as a value of type μ_2. Indeed, μ_1 and μ_2 represent the same sets of values, which we write $\mu_1 = \mu_2$. Richer types can be constructed using *type variables*. In type μ_3, X represents an arbitrary type. Informally, a message of type μ_3 must contain a label a, but the type of the associated value is not specified: the `pre` constructor allows us to impose the presence of a given field, even if its type is unspecified. Similarly, in μ_4, Y is a *field* variable. It can be either `abs`, $\mathtt{pre}(A)$ for any type A, or $\mathtt{pre}(X)$ for any type variable X. Finally, in μ_5, Z is a *row* variable that represent either `abs` or any list of fields. Note that we have $\mu_5 = \mu_6$. Remark also that some syntactically correct types, such as μ_7 and μ_8, can be meaningless: in particular labels must not occur twice, and a variable cannot have two different sorts (here X is used both as a field variable with label a, and as a row variable).

A component has a set of *server ports* and *client ports*. Each port is characterized by its name, and the type of the values it can carry. The type of a component is essentially a polymorphic function type. Figure 5 gives examples of components and component types. *id* has a polymorphic type. Its client and server ports can be used with any type X. *dup* duplicates its arguments. add_a adds a new field with label a to the messages it receives on client port i. Note that these messages must not contain label a. $remove_a$ removes the field named a, that may or may not be present. *reset* reset the value associated to label a to some initial value. *serialize* gets an arbitrary message $\{X\}$ and returns a new message with one field which is the serialized form of $\{X\}$. *deserialize* is the converse operation.

$$
\begin{aligned}
id &: \forall X.\{i:\{X\}\} \to \{o:\{X\}\}\\
dup &: \forall X.\{i:\{X\}\} \to \{o_1:\{X\}; o_2:\{X\}\}\\
add_a &: \forall X.\{i:\{a:\mathtt{abs}; X\}\} \to \{o:\{a:\mathtt{pre}(\mathtt{A}); X\}\}\\
remove_a &: \forall X,Y.\{i:\{a:Y; X\}\} \to \{o:\{a:\mathtt{abs}; X\}\}\\
reset &: \forall X.\{i:\{a:\mathtt{pre}(\mathtt{A}); X\}\} \to \{o:\{a:\mathtt{pre}(\mathtt{A}); X\}\}\\
serialize &: \forall X.\{i:\{X\}\} \to \{o:\{s:\mathtt{ser}(\{X\}); \mathtt{abs}\}\}\\
deserialize &: \forall X.\{i:\{s:\mathtt{ser}(\{X\}); \mathtt{abs}\}\} \to \{o:\{X\}\}
\end{aligned}
$$

Figure 5: Examples of component types

As for message types, some component types are meaningless. Consider the following type:

$$\forall X.\{i:\{a:\mathtt{pre}(X); \mathtt{abs}\}\} \to \{o:\{a:X\}\}$$

The two occurrences of X are used with a different meaning. The first one is a type variable whereas the second one is a field variable. In a more subtle way, the following type is incorrect:

$$\forall X.\{i:\{X\}\} \to \{o:\{a:\mathtt{pre}(\mathtt{A}); X\}\}$$

Both occurrences of X are row variable. However, the first one includes rows that may contain a field with label a, whereas the second does not.

Figure 6 depicts the same architecture definition as in figure 3, using these more precise types. The definition will be well-typed if and only if we can solve the equations:

$$
\begin{aligned}
\{X\} &= \{ts:\mathtt{pre}(\mathtt{A}); Y\}\\
\{X\} &= \{ts:\mathtt{abs}; Z\}
\end{aligned}
$$

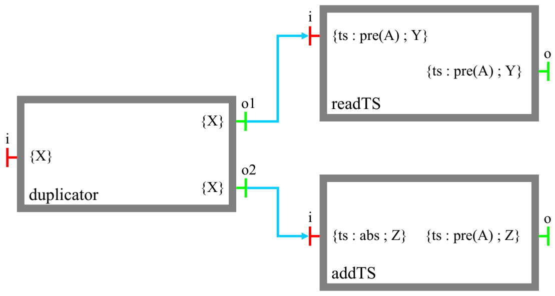


Figure 6: Example 1 revisited

Remark that we have chosen different type variables for each component. The equations do not have any solution, and thus the system is not well typed.

3.2 Formally

3.2.1 *Syntax*

We first introduce the syntax of messages types, which are very similar to record types.

$$\begin{array}{rll} \tau ::= \mu \mid \mathtt{ser}(\tau) \mid \sigma_B \mid \alpha & & \text{types} \\ \mu ::= \{\rho^{\emptyset}\} & & \text{message types} \\ \rho^L ::= \xi^L \mid \mathtt{abs}^L \mid a:\phi;\rho^{L\uplus\{a\}} & & \text{row} \\ \phi ::= \theta \mid \mathtt{abs} \mid \mathtt{pre}(\tau) & & \text{fields} \\ \sigma_B ::= \mathtt{A} \mid \mathtt{B} \mid \ldots & & \text{base types} \end{array}$$

A type τ may either be a message type μ, a serialized type $\mathtt{ser}(\tau)$, a base type σ_B, or a type variable α. A message type μ is a record type, described by a row $\rho^{\emptyset}$. We suppose that $a, b, c, \ldots$ range over a denumerable set of message labels L_m and L over finite subsets of L_m. Intuitively, a row ρ^L must not contain any field whose label is in L_m. In the case of a message type, there is no restriction as to which labels may occurs, hence the $\emptyset$ superscript.

A row ρ may either be a row variable ξ, the empty row $\mathtt{abs}$, or the concatenation of a field $a : \phi$ with a row where label a does not occur. This restriction is enforced by the use of the L superscript. For instance $\{a : \theta; (a : \theta'; \xi^L)\}$ is not syntactically correct. The $\uplus$ operator denotes disjoint union, it is only defined for disjoint sets.

The presence information ϕ is either a field variable θ, the indication that the field is absent $\mathtt{abs}$, or that it is present and carries a value of type τ, denoted $\mathtt{pre}(\tau)$.

Finally, σ_B range over base types, corresponding to Java types in Dream. We often write $\{a : \phi; b : \phi'; \xi^L\}$ for $\{a : \phi; (b : \phi'; \xi^L)\}$.

We next give the syntax of component types.

$$\begin{array}{ll} C ::= \forall\widetilde{\alpha}\widetilde{\theta}\widetilde{\xi^L}.\{I^{\emptyset}\} \to \{I^{\emptyset}\} & \text{Component} \\ I ::= i : \mu; I \mid \emptyset & \text{Interface Set} \end{array}$$

We define L_p as a denumerable set of *ports*, or interface names, ranged over by i,o and their decorated variants. A component type is composed of a set of *input* interfaces and a set of *output* interfaces. An interface consists of a port and the type of values exchanged on this port. We suppose ports to be distinct in a given interface set (input or output). We write $\widetilde{x}$ for a finite set of variables, and require that every (type, field, or row) variable be bound in the $\forall$ prefix of the component type.

In the previous subsection, we used the same syntactic category for type, row, and field variables and we omitted the superscripts on rows. The reason is that the sorts of variables and the superscripts can be automatically inferred. For instance, the type $\forall X.\{i : \{X\}\} \to \{o : \{a : \mathtt{pre}(\mathtt{A}); X\}\}$ is incorrect because it cannot be rewritten as $\forall\xi^L.\{i : \{\xi^L\}\} \to \{o : \{a : \mathtt{pre}(int); \xi^L\}\}$: L should be $\emptyset$ in the first occurrence of ξ and $\{a\}$ in the second one.

An *architecture definition* D is given by a list of component names and their type, and a list of connections between ports. For both syntactic categories, we let ϵ denote an empty list (of components or connections). We let c and its decorated variants range over L_c, a denumerable set of *component names.*

$$\begin{array}{lll} D ::= (Cp, Co) & & \text{Architecture Definition} \\ Cp ::= \epsilon \mid c : C, Cp & & \text{Components} \\ Co ::= \epsilon \mid c.o = c'.i, Co & & \text{Connections} \end{array}$$

An architecture definition (Cp, Co) is well-formed if

- Component names in Cp are pairwise distinct.
- For every connection $c.o = c'.i$ in Co, $c : C$ and $c' : C'$ are in Cp for some C, C'. Moreover, o is a client port (i.e., it is a port of the output set of interfaces) of C and i a server port of C'.
- Any port is connected at most once.

3.2.2 *Typing*

We write T the set of rows ρ^L for all L. We define an equational theory E on T with the following axioms and rule.

$$a : \phi; (a' : \phi'; \rho^L) = a' : \phi'; (a : \phi; \rho^L)$$
$$a : \phi; \mathtt{abs}^L = a : \phi; (b : \mathtt{abs}; \mathtt{abs}^{L\uplus\{b\}})$$
$$a : \phi; \xi^L = a : \phi; (b : \theta; \xi'^{L\uplus\{b\}})$$
$$\rho^L = \rho'^L \implies a : \phi; \rho^{L\uplus\{a\}} = a : \phi; \rho'^{L\uplus\{a\}}$$

The first axiom states that the order in the definition of the fields does not matter. The second states that the $\mathtt{abs}$ row denotes rows containing only absent fields. The third axiom states that a row variable denotes rows with fields whose presence information is not specified.

We know from [16] that the problem of unification in T modulo E is decidable and syntactic: every solvable unification problem has a most general unifier.

From an architecture definition $D = (Cp, Co)$, we can generate a set of equations $E(D)$. First we get a list Cp' by suppressing all quantifiers in Cp, assuming variables are first renamed such that no variable appear in two distinct types. We write $Cp'(c)$ for the type of component c in Cp'. For a component type C, we note $T_C(C.o)$ for the type associated with client interface o. We define similarly $T_S(C.i)$.Using these definitions, we can define E as follows:

$$E(Cp, Co) = \{T_C(Cp'(c).o) = T_S(Cp'(c).i) \mid c.o = c.i \in Co\}$$

An architecture definition D is typable if and only if E admits an unifier.

3.3 Example

Figure 7 (a) depicts a stack of dream components. The component *producer* at the top of the left stack generates messages consisting of a unique chunk of type `TestChunk` and name tc.

$$producer : \{\} \to \{o : \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}\}$$

The component *serializer* returns messages with a unique chunk sc that is the serialized form of the messages received on input port i.

$$serializer : \forall X.\{i : \{X\}\} \to \{o : \{sc : \mathtt{ser}(\{X\}); \mathtt{abs}\}\}$$

Component *addIP* adds a chunk of type `IPChunk` and name *ipc* to a message that does not contain an *ipc* chunk.

$$addIP : \forall X.\{i : \{ipc : \mathtt{abs}; X\}\} \to \{o : \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\}\}$$

channelOut dispatches messages on the network, and requires them to define at least an *ipc* chunk of type `IPChunk`.

$$channelOut : \forall X.\{i : \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\}\} \to \{o : \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\}\}$$

The right stack performs the symmetric actions. Figures 7 (b) and (c) show two incorrect architectures. In (b), the deserializer component is missing and in (c) the deserializer and addIP components are inverted. Architecture (a) is well-typed but (b) and (c) are not. Consider architecture (b), we deduce the following equations from the linking (note that bound variables have been renamed).

$$\{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\} = \{U\} \quad (1)$$
$$\{sc : \mathtt{pre}(\mathtt{ser}(U)); \mathtt{abs}\} = \{ipc : \mathtt{abs}; Z\} \quad (2)$$
$$\{ipc : \mathtt{pre}(\mathtt{IPChunk}); T\} = \{ipc : \mathtt{pre}(\mathtt{IPChunk}); Z\} \quad (3)$$
$$\{ipc : \mathtt{pre}(\mathtt{IPChunk}); Z\} = \{Y\} \quad (4)$$
$$\{Y\} = \{ipc : \mathtt{pre}(\mathtt{IPChunk}); X\} \quad (5)$$
$$\{ipc : \mathtt{abs}; X\} = \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\} \quad (6)$$

From 6, we deduce that

$$X = \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}$$

Then from 5, we have

$$Y = \{ipc : \mathtt{pre}(\mathtt{IPChunk}); tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}$$

It follows from 4 and 3 that

$$T = Z = \{tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}\}$$

Besides, we deduce from 2 that

$$Z = \{sc : \mathtt{pre}(\mathtt{ser}(U)); \mathtt{abs}\}$$

The terms $tc : \mathtt{pre}(\mathtt{TestChunk}); \mathtt{abs}$ and $sc : \mathtt{pre}(\mathtt{ser}(U)); \mathtt{abs}$ are obviously not unifiable and thus the system is not typable.

We implemented this type system in Ocaml. It takes as input an architecture definition, checks that it is well-sorted, generates a system of equations and try to solve it. We used this tool to check the validity of several assemblages. Figure 8 corresponds to the input file for architecture (c).

Our prototype fails to solve the equations corresponding to this architecture. The output corresponds to a set of equations equivalent to the initial system, when the unification algorithm encounters a contradictory equation (*e.g.*$\mathtt{abs} = \mathtt{IPChunk}$).

```
producer:
  {}->{o:{tc:pre(TestChunk);abs}}
consumer:
  {i:{tc:pre(TestChunk);abs}}->{}
serializer:
  {i:{'x}}->{o:{s:pre(ser({'x}));abs}}
deserializer:
  {i:{s:pre(ser({'x}));abs}}->{o:{'x}}
addIP:
  {i:{ipc:abs;'x}}->{o:{ipc:pre(IPChunk);'x}}
removeIP:
  {i:{ipc:pre(IPChunk);'x}}->{o:{ipc:abs;'x}}
channelOut:
  {i:{ipc:pre(IPChunk);'x}}->{o:{ipc:pre(IPChunk);'x}}
channelIn:
  {i:{'x}}->{o:{'x}}
composite c is
  {}->{}
with
  producer.o = serializer.i
  serializer.o = addIP.i
  addIP.o = channelOut.i
  channelOut.o = channelIn.i
  channelIn.o = deserializer.i
  deserializer.o = removeIP.i
  removeIP.o = consumer.i
end
```

Figure 8: file `archC.d`

```
% dtype archC.d
No solution
-----------
abs = IPChunk
abs = abs
{tc:TestChunk;abs} = {ipc:IPChunk;'removeIP_x}
{ipc:abs;'removeIP_x} = {tc:TestChunk;abs}
```

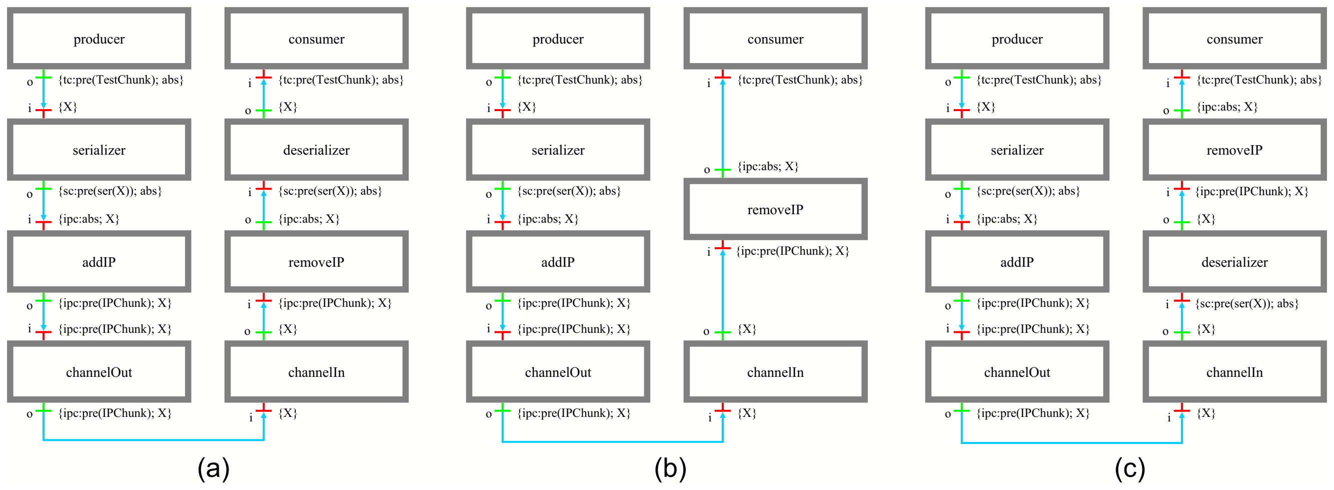


Figure 7: Example: a stack of components

3.4 Discussion and limitations

The main limitation is that this typing discipline is too restrictive to type certain Dream components. Typically, they can exhibit different behavior depending on the presence of a given label in a message (*e.g.*, routers). Consider for instance a component `route` that routes messages it gets on its client port on different server port depending on the presence of a given label. We would like its type to be something like:

$$\begin{aligned}\texttt{route} : \forall X.\ &\{i : \{a : \texttt{pre}(\texttt{A}); X\}\} \to \{o_1 : \{X\}; o_2 : \{\texttt{abs}\}\} \\ \wedge\ \ &\{i : \{a : \texttt{abs}; X\}\} \to \{o_1 : \{\texttt{abs}\}; o_2 : \{X\}\}\end{aligned}$$

Similarly, some components may output messages of different types.

$$\begin{aligned}\texttt{produce} : &\{\} \to \{o : \{a : \texttt{abs}; b : \texttt{B}; \texttt{abs}\}\} \\ \wedge &\{\} \to \{o : \{a : \texttt{A}; b : \texttt{abs}; \texttt{abs}\}\}\end{aligned}$$

In both cases, we can find approximating types that allow us to type a definition involving these components. For instance:

$$\begin{aligned}\texttt{route} : \forall XYZ.\quad &\{i : \{X\}\} \to \{o_1 : \{Y\}; o_2 : \{Z\}\} \\ \texttt{produce} : \forall XY.\quad &\{\} \to \{o : \{a : X; b : Y; \texttt{abs}\}\}\end{aligned}$$

In doing so, we lose any guarantee about the correctness of the architecture definition, since obviously, the code of the components does not conform to these types.

4. RELATED WORK

The type system presented in this paper constitutes an example of a domain specific type system, tailored to checking architectural constraints in the component-based Dream environment. Type systems that capture various properties of programs have of course been intensively studied for various languages, including ML, Java, as well as in more abstract settings such as process algebras and the π-calculus [18]. Exploiting type systems for checking architectural constraints has received less attention, but has nevertheless been the subject of various works in the past decade. We can mention for instance work on the Wright language [3], which supports the verification of behavioral compatibility constraints in a software architecture, matching a component with a role; recent work on ArchJava [2], which uses ownership types to enforce communication integrity in a Java-based component model; and more recent work on behavioral contracts for component assembly [6]. The type systems (or compatibility relations) used in these works, however, do not capture the architectural constraints that are dealt with in this paper. Both the Wright system and the behavioral contract system would need to be extended to deal with the record types that characterize Dream messages, and the ArchJava type system is tailored to enforce communication integrity, *i.e.* , to prevent aliasing that may destroy a component integrity. The work which is closest to ours is probably the recent work on the type system for the Ptolemy II system [13], which combines a rich set of data types, including structured types such as (immutable) arrays and records, and a behavioral type system which extends the work on interface automata [7, 8] for capturing temporal aspects of component interfaces. The Ptolemy II type system would not be directly applicable to our Dream constraints, though, for it features only immutable record types. However, a combination of extensive record types as in this paper and behavioral types of the Ptolemy II system is definitely worth investigating.

5. CONCLUSION AND FUTURE WORK

We have presented a domain specific type system for messages and components that manage messages in the Dream framework. This type system is based on existing work on extensible records, and is rich enough to address component assemblages such as protocol stacks, as illustrated in Section 3.3.

An obvious shortcoming of our approach is that we do not formally state the guarantees provided by the type system, namely that there will be no run-time error due to the access of an absent message chunk, the addition of a chunk whose name is already present in the message, or the use of a chunk's contents at a wrong type. We have taken the more pragmatic approach of first implementing and testing the expressivity of the type system. We plan on formalizing the behavior of Dream components and state the guarantees of our type system as continuation of this work.

Experimenting with our type system has shown that it is not precise enough when the behavior of a component depends on the structure of a message, as described in Section 3.4. To address this issue, we plan on adapting existing works on intersection types, such as [15], to our setting.

Finally, we are studying the integration of our type checking phase in the Dream ADL processing workflow.

6. REFERENCES

[1] Enterprise JavaBeansTM Specification, Version 2.1, August 2002. Sun Microsystems, `http://java.sun.com/products/ejb/`.

[2] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings ECOOP 2002, LNCS 2548*. Springer, 2002.

[3] 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.

[4] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, , and K. Saikoski. The Design and Implementation of Open ORB v2. In *IEEE Distributed Systems Online Journal, vol. 2 no. 6*, November 2001.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland, 2004.

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

[7] L. de Alfaro and T. Henzinger. Interface Automata. In *Proceedings of the joint 8th European software engineering conference and 9th ACM SIGSOFT international symposium on the foundations of software engineering (ESEC/FSE 01)*, 2001.

[8] L. de Alfaro and T. Henzinger. Interface Thoeries for Component-Based Design. In *Proceedings of EMSOFT '01*, volume 2211 of *Lecture Notes in Computer Science*. Springer, 2001.

[9] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *USENIX Annual Technical Conference*, 2002.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *SOSP'97*, 1997.

[11] F. Kon, T. Yamane, K. Hess, R. H. Campbell, and M. D. Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, USA, January 2001.

[12] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs. In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM'2004)*, Toronto, Canada, October 2004.

[13] E. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3), 2004.

[14] P. Merle, editor. *CORBA 3.0 New Components Chapters*. OMG TC Document ptc/2001-11-03, November 2001.

[15] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

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

[17] D. Rémy. Typing record concatenation for free. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[18] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.