DREAM Types

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

Philippe Bidinger[1], Matthieu Leclercq[1], Vivien Quéma[1,2], Alan Schmitt[1], Jean-Bernard Stefani[1]

[1]INRIA, France

[2]Institut National Polytechnique de Grenoble, France

SAVCBS 2005

Outline

1. Motivations
   - Component-based programming
   - The DREAM framework
   - Problem statement
2. DREAM types
   - Overview
   - Message types
   - Component types
   - Checking a configuration
3. Use case
4. Conclusion

Outline

1. Motivations
   - Component-based programming
   - The DREAM framework
   - Problem statement
2. DREAM types
   - Overview
   - Message types
   - Component types
   - Checking a configuration
3. Use case
4. Conclusion

Component-based programming

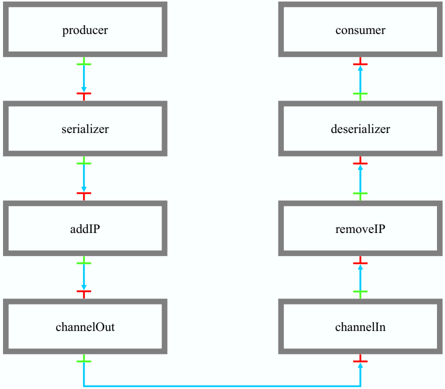
- Component-based frameworks have emerged in the past two decades:
  - applications (EJB, CCM)
  - middleware (dynamicTAO, OpenORB)
  - operating systems (OSKit, THINK)
- A component:
  - is independently deployable
  - is configurable (attributes)
  - has interfaces (client, server)
  - communicate through bindings between interfaces

Outline

The DREAM framework

- Component framework for constructing message-oriented middleware (MOM)
  - General component model
  - Component library
    - Message queues, serializer, channels, routers, . . .
  - Tools for the description, configuration and deployment of MOMs
- Various MOMs can be built:
  - Publish/Subscribe, Event/Reaction, Group communication protocols, . . .

A simplistic DREAM MOM



DREAM messages

- DREAM components exchange messages
  - Messages are Java objets that encapsulate named chunks
  - Each chunk implements an interface that defines its type
- Basic operations over messages
  - read, add, remove, or update a chunk of a given name

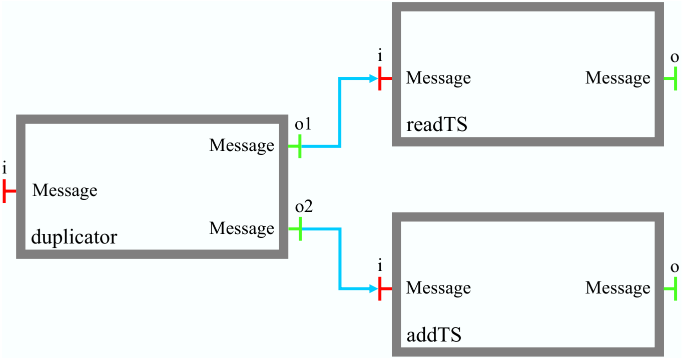
Outline

Problem statement

- Three kinds of run-time errors
  - A chunk is absent when it should be present
  - A chunk is present when it should be absent
  - A chunk does not have the expected type

- But... all messages in DREAM have the same type: the `Message` Java interface

Example


i
Message
duplicator
o1
Message
o2
Message
i
Message
Message
o
readTS
i
Message
Message
o
addTS

Outline

Overview

Goals

Catching configurations errors early on, when writing the architecture description of a DREAM MOM

How?

By defining a richer type system allowing the description of:

- the internal structure of messages
- the behavior of components

Overview

Goals

Catching configurations errors early on, when writing the architecture description of a DREAM MOM

How?

By defining a richer type system allowing the description of:

- the internal structure of messages
- the behavior of components

Type system

- Adaption of existing work on type systems for extensible records for ML (D. Rémy, 1993)

Definition

An extensible record is a finite set of associations, called *fields*, between labels and values

- DREAM messages can be seen as records, where each chunk correspond to a field of the record
- DREAM components can be seen as polymorphic functions

Type system

- Adaption of existing work on type systems for extensible records for ML (D. Rémy, 1993)

Definition

An extensible record is a finite set of associations, called *fields*, between labels and values

- DREAM messages can be seen as records, where each chunk correspond to a field of the record
- DREAM components can be seen as polymorphic functions

Type system

- Adaption of existing work on type systems for extensible records for ML (D. Rémy, 1993)

Definition

An extensible record is a finite set of associations, called *fields*, between labels and values

- DREAM messages can be seen as records, where each chunk correspond to a field of the record
- DREAM components can be seen as polymorphic functions

Outline

Message types

- A message type consists of:
  - a list of pairwise distinct labels together with
    - the type of the corresponding value
    - a special tag `abs` if the message does not contain the given label
- Includes *type*, *field*, and *row* (record) variables
- `ser`, an ad-hoc type constructor
  - if τ is an arbitrary type, $\mathtt{ser}(\tau)$ is the type of serialized values of type τ

Examples

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

Outline

Component types

- A component has a set of *server ports* and *client ports*
- Each port is characterized by:
  - its name
  - the type of the values it can carry
- The type of a component is polymorphic, mapping client port types to server port types
- Polymorphism is important for two reasons:
  - the same component can be used in different contexts with different types
  - it expresses explicit dependencies between client and server port types

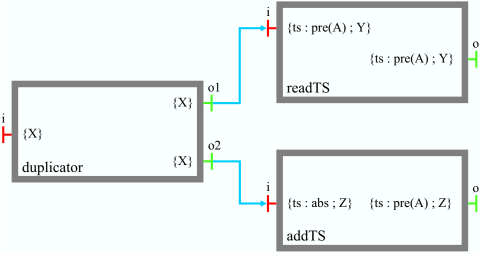
Examples

$$\textbf{id} : \forall X.\{i : \{X\}\} \to \{o : \{X\}\}$$
$$\textbf{dup} : \forall X.\{i : \{X\}\} \to \{o_1 : \{X\}; o_2 : \{X\}\}$$
$$\textbf{add}_\textbf{a} : \forall X.\{i : \{a : \mathtt{abs}; X\}\} \to \{o : \{a : \mathtt{pre}(\mathrm{A}); X\}\}$$
$$\textbf{remove}_\textbf{a} : \forall X, Y.\{i : \{a : Y; X\}\} \to \{o : \{a : \mathtt{abs}; X\}\}$$
$$\textbf{reset} : \forall X.\{i : \{a : \mathtt{pre}(\mathrm{A}); X\}\} \to \{o : \{a : \mathtt{pre}(\mathrm{A}); X\}\}$$
$$\textbf{serialize} : \forall X.\{i : \{X\}\} \to \{o : \{s : \mathtt{ser}(\{X\}); \mathtt{abs}\}\}$$
$$\textbf{deserialize} : \forall X.\{i : \{s : \mathtt{ser}(\{X\}); \mathtt{abs}\}\} \to \{o : \{X\}\}$$

Outline

Checking a configuration

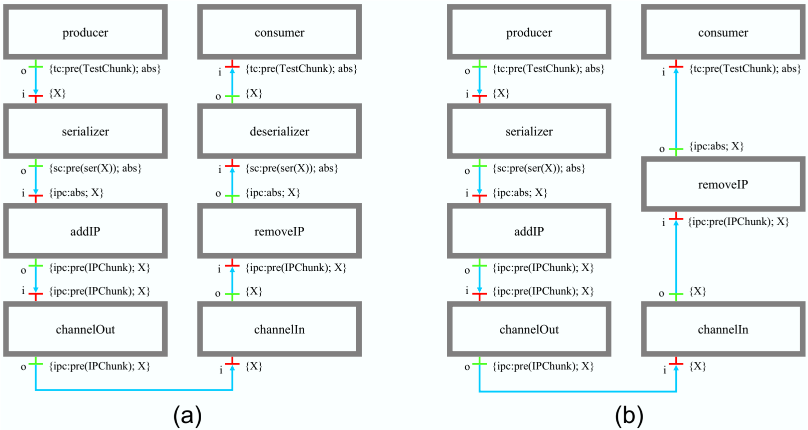
- Type checking using equational theory and unification algorithm (D. Rémy, 1993)



Configuration well-typed iff we can solve the equations:

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

Use case


producer
o {tc:pre(TestChunk); abs}
i {X}
serializer
o {sc:pre(ser(X)); abs}
i {ipc:abs; X}
addIP
o {ipc:pre(IPChunk); X}
i {ipc:pre(IPChunk); X}
channelOut
o {ipc:pre(IPChunk); X}
consumer
i {tc:pre(TestChunk); abs}
o {X}
deserializer
i {sc:pre(ser(X)); abs}
o {ipc:abs; X}
removeIP
i {ipc:pre(IPChunk); X}
o {X}
channelIn
i {X}
(a)
producer
o {tc:pre(TestChunk); abs}
i {X}
serializer
o {sc:pre(ser(X)); abs}
i {ipc:abs; X}
addIP
o {ipc:pre(IPChunk); X}
i {ipc:pre(IPChunk); X}
channelOut
o {ipc:pre(IPChunk); X}
consumer
i {tc:pre(TestChunk); abs}
o {ipc:abs; X}
removeIP
i {ipc:pre(IPChunk); X}
o {X}
channelIn
i {X}
(b)

Use case

From bindings, we deduce the following equations:

$$\{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\} = \{U\} \tag{1}$$

$$\{sc : \texttt{pre}(\texttt{ser}(U)); \texttt{abs}\} = \{ipc : \texttt{abs}; Z\} \tag{2}$$

$$\{ipc : \texttt{pre}(\texttt{IPChunk}); T\} = \{ipc : \texttt{pre}(\texttt{IPChunk}); Z\} \tag{3}$$

$$\{ipc : \texttt{pre}(\texttt{IPChunk}); Z\} = \{Y\} \tag{4}$$

$$\{Y\} = \{ipc : \texttt{pre}(\texttt{IPChunk}); X\} \tag{5}$$

$$\{ipc : \texttt{abs}; X\} = \{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\} \tag{6}$$

Use case

From 6, we deduce that

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

Then from 5, we have

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

It follows from 4 and 3 that

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

Besides, we deduce from 2 that

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

$tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}$ and $sc : \texttt{pre}(\texttt{ser}(U)); \texttt{abs}$ are not unifiable $\Rightarrow$ the configuration is not correct

Conclusion

- Domain specific type system for messages and components
  - Based on existing work on extensible records
  - Rich enough to address component assemblages such as protocol stacks
- FFS: type system is too restrictive to type DREAM components that exhibit different behavior depending on the presence of a given label in a message (e.g. routers)
  - DREAM operational semantics
  - Intersection types

For Further Reading I

M. Leclercq, V. Quéma and J.-B. Stefani.

DREAM: *a Component Framework for Constructing Resource-Aware, Configurable Middleware*.

IEEE Distributed Systems Online, vol. 6 no. 9, 2005.

E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma and J.-B. Stefani.

FRACTAL: *an Open Component Model and its Support in Java*.

Proceedings of the International Symposium on Component-based Software Engineering (CBSE), 2004.

For Further Reading II

P. Bidinger, A. Schmitt and Jean-Bernard Stefani.
An Abstract Machine for the Kell Calculus.
Proceedings of the International Conference on Formal Methods for Object-Based Distributed Systems (FMOODS), 2005.

D. Hirschkoff, T. Hirschowitz, D. Pous, A. Schmitt and J.-B. Stefani.
Component-Oriented Programming with Sharing: Containment is not Ownership.
Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), 2005.

Questions

- `http://dream.objectweb.org` – DREAM implementation and documentation
- `http://sardes.inrialpes.fr/kells` – Kell calculus papers and implementation