Distributed deployment with Oz

J-B. Stefani & C. Taton

INRIA

May 2008

Plan

1. **Introduction**
2. **Building and manipulating dynamic architectures**
   - FructOz in more detail
   - Deployment examples
3. **Navigating dynamic architectures**
   - Navigation primitives
   - Dynamic FPath
4. **Evaluation**
5. **Conclusion**

Goal

Dynamic software architectures

Describing dynamic architectures

- Structure
- Dynamicity

Manipulating dynamic architectures

- Naming and navigating within architectures
- Monitoring and supervision

Motivations

- Limitations in the Jade/Java implementation
- Experimenting self-optimisation and self-configuration

Motivations

- Support for complex deployment & configuration processes
- Deployment & configuration process = distributed workflow
- Configuration = module resolution and architecture construction

Contributions

FructOz An Oz framework for deploying and configuring complex dynamic software architectures

LactOz An Oz library for navigating and monitoring dynamic software architectures

FructOz
Building and manipulating dynamic architectures

FructOz : Objectives

Construction Deployment and configuration processes

Manipulation Internalizing (re)deployment and (re)configuration

- Synchronizations (workflow)
- Lazy deployment
- Exception and fault handling
- Parametric architectures
- Dynamic architectures

- Lightweight Fractal implementation
- Exploiting the Mozart/Oz environment

Micro tutorial Oz (1)

Procedures

```
local
    proc {MyProc Param1 Param2 ?Param3}
        Param3 = Param1 + 2*Param2
    end
    P1 = 1
    P2 = 2
    P3
in
    {MyProc Param1 Param2 Param3}
end
```

Thread

```
thread
    %% Do whatever you want here!
end
```

Micro tutorial Oz (2)

Lists and streams

Lists

```
ANiceList = [1 2 3] = 1 | 2 | 3 | nil % create a finite list
MaybeAnInfiniteList = 1 | 2 | _ % create a list whose tail is undefined yet

for I in AnotherList do % iterate over a list
    {Show I}
end
```

Streams

```
Stream Port
{NewPort Stream Port} % create a port: Stream = _

{Send Port 1} % append new messages to the stream: Stream = 1 | _
{Send Port 2} % Stream = 1 | 2 | _
{Send Port 3} % Stream = 1 | 2 | 3 | _
```

Micro tutorial Oz (3)

Synchronization

Locks

```
L = {NewLock} % Create a lock
lock L then
    %% Mutual exclusion
end
```

Variable binding

```
local
    X % X exists but is free (i.e. unbound, undetermined)
in
    if {IsDet X} then ... end
    ...
    {Wait X} % Synchronize on X being defined (suspend the current thread)
end
```

Micro tutorial Oz (3)

Synchronization

By-need synchronization

```
X = {ByNeed
        fun {$}
        ... % What will be X' value
        end}
%% X undetermined here
...
if {IsNeeded X} then ... end
...
%% Synchronize on X value being (or having been) requested by another thread
{WaitQuiet X}
...
%% Make X needed
Y = X + 1
{MakeNeeded X}
```

FructOz under the microzcope

Navigable entities

Interface Reconfiguration point

Binding Link between interfaces

Membrane Set of interfaces

- Navigation using tags (filtering)
- Implicit composition
- No special support for controllers

Primitives

Interface

INew : *kind* $\times$ *tags* $\rightarrow \mathcal{I}$, creation

IImplements : $\mathcal{I} \times$ *native* $\rightarrow$ *unit*, implementation definition

IResolveSync : $\mathcal{I} \rightarrow$ *native*, synchronous proxy

IResolveAsync : $\mathcal{I} \rightarrow$ *native*, asynchronous proxy

Liaison (Binding)

BNew : $\mathcal{I} \times \mathcal{I} \rightarrow \mathcal{B}$, creation

BBreak : $\mathcal{B} \rightarrow$ *unit*, destruction

Membrane (Component)

CNew : *tags* $\rightarrow \mathcal{C}$, creation

CAddInterface : $\mathcal{C} \times \mathcal{I} \rightarrow$ *unit*, interface addition

CRemoveInterface : $\mathcal{C} \times \mathcal{I} \rightarrow$ *unit*, interface removal

Component package

Oz Functor

- Container (code & data)
- Instantiation procedure (i.e. deployment)
- Module : instance of functor (export)
- Functor imports (dependencies)
- Module factory : ModuleManager (import resolution)
- Invocation : NewModule = MM apply(MyFunctor $)

Distributed environment

- Remote factory : RemoteModuleManager
- Invocation : NewModule = RMM apply(MyFunctor $)

Component package

Usage

```
functor MyFunctor
import System
export Pi Fibo
define
    {System.showInfo "Initializing my very cool module..."}
    Pi = 3.14159
    fun {Fibo N} if (N =< 1) then 1 else {Fibo (N-1)} + {Fibo (N-2)} end
end
```

Component package

Usage (core language)

```
proc {MyFunctor LImports ?Export}
    [System] = LImports
    Pi
    Fibo
in
    {System.showInfo "Initializing␣my␣very␣cool␣module..."}
    Pi = 3.14159
    fun {Fibo N} if (N =< 1) then 1 else {Fibo (N−1)} + {Fibo (N−2)} end

    Export = 'export'(pi:Pi fibo:Fibo)
end
```

Component package

Defining components

```
functor MyCompPackage
export Membrane
define
    Comp = {CNew} % Create an empty membrane

    Itf = {INew server [mytag]} % Create a server interface
    {IImplements Itf ...} % Binds the interface to some implementation

    {CAddInterface Comp Itf} % Register the interface in the membrane

    Membrane = Comp % Release and export the component (its membrane)
end
```

Composition

- Additional layer (Fractal controllers)
- Composition contexts
- Distribution context

Primitives

CAddSubComponent : $\mathcal{C} \times \mathcal{C} \times \textit{context} \rightarrow \textit{unit}$

CRemoveSubComponent : $\mathcal{C} \times \mathcal{C} \times \textit{context} \rightarrow \textit{unit}$

CGetSubComponent : $\mathcal{C} \times \textit{context} \rightarrow \mathcal{S}\{\mathcal{C}\}$

CListCompositionContexts : $\mathcal{C} \rightarrow \mathcal{S}\{\textit{context}\}$

Deployment examples

Parameterized distributed deployment

Capabilities

- Distributed deployment
- Workflow patterns
- Parameterization

Deployment scenarios

1. Centralized sequential
2. Centralized parallel
3. Distributed hierarchical

Parameterized distributed deployment

Simple Composite

```
fun {MyWonderfulCompositePackage Cluster N DeployProc}
   functor $
   export Membrane
   define
      Comp = {CNew}
      {DeployProc MySubcomponentPackage Cluster N Comp}
      Membrane = Comp
   end
end
```

- Parameterized component

Parameterized distributed deployment

Centralized sequential

```
proc {DeploySeq CompPackage Cluster N Parent}
    NextRoundRobin = {MakeRoundRobin Cluster}
    for I = 1..N do
        Host = {NextRoundRobin}
        NewComp = {RemoteDeploy Host CompPackage}
        {CAddSubComponent Parent NewComp}
    end
end
```

Parameterized distributed deployment

Centralized parallel asynchronous

```
proc {DeploySeq CompPackage Cluster N Parent}
    NextRoundRobin = {MakeRoundRobin Cluster}
    for I = 1..N do
        thread
            Host = {NextRoundRobin}
            NewComp = {RemoteDeploy Host CompPackage}
            {CAddSubComponent Parent NewComp}
        end
    end
end
```

Parameterized distributed deployment

Barrier synchronization

```
%%% Barrier synchronization
%%% LProcs: list of procedures
proc {Barrier LProcs}
   %% Create an array of unbound variables.
   Sync = {Tuple.make sync {List.length LProcs}}
   I = {NewCell 1}
in
   for Proc in LProcs do
      thread
         {Proc} % Execute the procedure in a separate thread.
         Sync.@I = true % Notify procedure termination.
      end
      I := @I + 1
   end
   %% Wait for all signals
   {Record.forAll Sync Wait}
end
```

Parameterized distributed deployment

Centralized parallel asynchronous

```
proc {DeployParallel CompPackage Cluster N Parent}
   NextRoundRobin = {MakeRoundRobin Cluster}
   %% Deployment script
   proc {DeployProc}
      Host = {NextRoundRobin}
      NewComp = {RemoteDeploy Host CompPackage}
      {CAddSubComponent Parent NewComp}
   end
   %% Generate a list of N deployment procedures [DeployProc DeployProc ... DeployProc]
   LDeployments = {MakeCopyList DeployProc N}
in
   %% Execute and synchronize on the scripts' execution
   {Barrier LDeployments}
end
```

Parameterized distributed deployment

Distributed hierarchical

```
NextRoundRobin = {MakeRoundRobin Cluster}
proc {DeployTree CompPackage Arity Depth Parent}
   functor DistributedDeployProc
   export Membrane
   define
      if (Depth > 0) then
         {DeployTree CompPackage Arity (Depth − 1) Parent}
      end
      Membrane = {Deploy CompPackage} % deploy component locally
   end
   proc {DeployProc}
      Host = {NextRoundRobin}
      NewComp = {RemoteDeploy Host DistributedDeployProc}
      {CAddSubComponent Parent NewComp}
   end
in
   {Barrier {MakeCopyList DeployProc Arity}}
end
```

Lazy deployment

Lazy instantiation

```
%% Lazily deployed implementation
Itf = {INew server [tags ...]}
{IImplements Itf
    {ByNeed fun {$} % Implementation will be lazily instantiated
        {New class $ ... end}}}
```

Lazy component instantiation

```
%% Lazily deployed component
Client = {ByNeed fun {$} {Deploy LazyClient} end}
```

Lazy deployment

Lazy binding instantiation

```
%% Lazily deployed binding between a Client component and a Server component
B = thread
      %% Wait until someone needs and thus triggers the deployment of the
      %% Client component. Once this has happened, get the client interface
      IFrom = {CGetInterface {WaitQuietValue Client} [client]}
      %% Create a lazy reference to the server interface
      ITo = {ByNeed fun {$} {CGetInterface Server [service]} end}
   in
      %% Create a lazy binding between Client (now determined) and Server
      %% (might still be lazy)
      {BNewLazy IFrom ITo}
   end
```

LactOz
Navigating dynamic architectures

LactOz

Library for dynamic architecture navigation

- Anchor points within FructOz architecture
- Distributed event bus (using Mozart/Oz infrastructure)
- Quasi transparency

Navigation primitives

Component

CGetInterfaces : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{I}\}$

Interface

IGetComponent : $\mathcal{I} \rightarrow \mathcal{C}$

IGetBindingsFrom : $\mathcal{I} \rightarrow \mathcal{S}\{\mathcal{B}\}$

IGetBindingsTo : $\mathcal{I} \rightarrow \mathcal{S}\{\mathcal{B}\}$

Binding

BGetClientInterface : $\mathcal{B} \rightarrow \mathcal{I}$

BGetServerInterface : $\mathcal{B} \rightarrow \mathcal{I}$

Dynamic FPath
Primitives

Set operations

SUnion, SIntersection, SSubtract

SFilter, SFilterHas(All/One)Tag(s)

SEmpty, SContains $: \mathcal{S} \rightarrow \mathbb{B}, \mathcal{S} \times \textit{any} \rightarrow \mathbb{B}$

Value operations

- Booleans (BNot, BAnd, BOr)
- Numerals (NSum, NSubtract, NMax, NMin, etc)

Example

Constructing a filter by composition

Definition...

$$\text{CGetExternalComponentsBoundFrom} : \mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$$

Given a component C, get all component C' such that there exists a binding from C to C'.

Exploring

1. Component C
2. Client interfaces of C
3. Client bindings of C
4. Server interfaces bound to client interfaces of C
5. Server components C' bound to C

Example

Constructing a filter by composition

Client interfaces of a component

```
fun {CGetClientInterfaces C}
    AllItfs = {CGetInterfaces C}
    %% The kind of an interface cannot change,
    %% so we optimize using a dynamic set filtering with a static filter function
    {SStaticFilter AllItfs (fun {$ I} I.kind == client end)}
end
```

Server component at the end of a binding

```
fun {BGetServerComponent B}
    {IGetComponent {BGetServerInterface B}}
end
```

Example

Constructing a filter by composition

Server components bound to a component

```
fun {CGetExternalComponentsBoundFrom C}
    ClientItfs = {CGetClientInterfaces C}
    ClientBindings = {SUnion {SMap ClientItfs (fun {$ I} {IGetBindingsFrom I} end)}}
    {SMap ClientBindings (fun {$ B} {BGetServerComponent B} end)}
end
```

Example

Constructing a filter by composition

Extracting a subset

Filter from these components those that have at least a subcomponent tagged “interesting”.

Implementation

```
{SFilter {CGetExternalComponentsBoundFrom C}
 fun {$ C}
    Children = {CGetSubComponents C Context}
 in
    {SIsEmpty {SFilterHasTag Children 'interesting'}}
 end}
```

Implementing dynamic architectures

Dynamic Variables

- Event-driven style
- Simple variables
  update(NewValue)
- Sets
  add(SElements)
  remove(SElements)

Example

Automatic binding

Dynamic architecture

```
%% Acquire a reference on the set of components we want to bind
%% from the Client component
SToBind = {SFilterHasTag {CGetSubComponents Comp content} 'tobound'}

%% Initializes and register a set listener as a synchronous listener
Listener = {New MySetListener init(SToBind)}
{SToBind listen(sync Listener)}
```

Example

Automatic binding

Observer

```
class MySetListener from SetListener
    meth init(Source)
        ...
    end

    meth add(SComps)
        {Set.forAll SComps
            proc {$ Comp}
                {BNew
                    {CGetInterface ServerComp client} % cheating here!
                    {CGetInterface Comp service}}
            end}
    end

    meth remove(SComps)
        ...
    end
end
```

Evaluation

Performance

Comparing technologies for a simple deployment

	Component deployment	Remote invocation
SmartFrog/Java RMI	295.5 ms	0.097 ms
Julia/Fractal RMI	159.5 ms	0.099 ms
FructOz	146.3 ms	0.0048 ms
FructOz/Julia Bridge	262.3 ms	N/A

TAB.: Comparison : FructOz, SmartFrog et Julia

Performance

Distributed deployment scalability

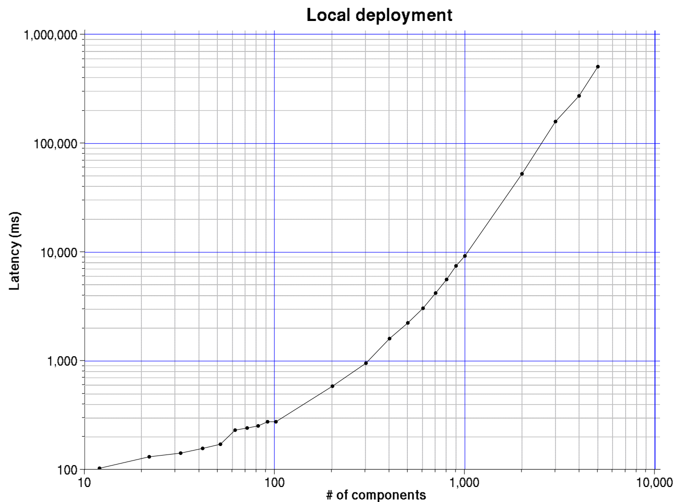


FIG.: Local deployment

Performance

Distributed deployment scalability

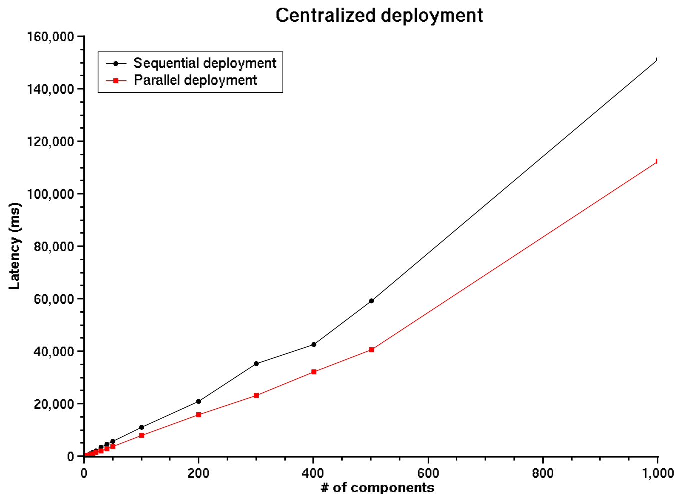
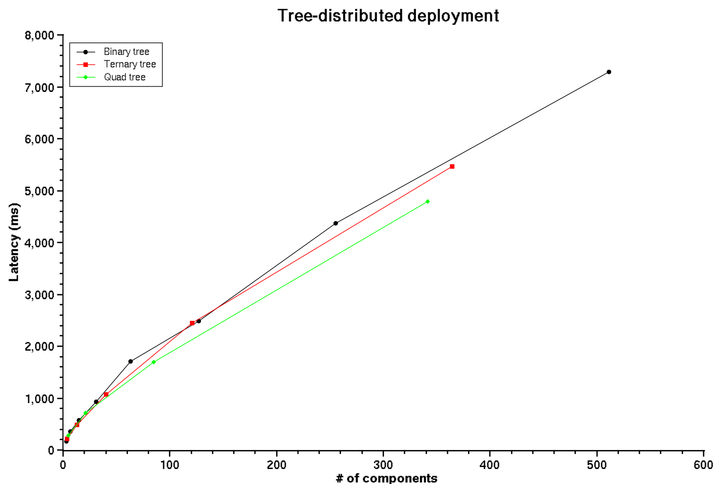


FIG.: Tree deployment evaluation

Performance

Distributed deployment scalability



Difficulties with Mozart/Oz infrastructure

- Garbage collection
- Marshalling
- Limitations on distributed environment
- Limited transparency (stateful entities)

What next ?

- Workflow patterns in Oz
- Automated monitoring (using LactOz)
- Atomic/compensatable actions for dependable/defeasible deployment/(re)configuration
- Exploiting & deploying overlays
  - slicing
  - publish/subscribe for event notifications and monitoring
  - probabilistic guarantees
  - etc

Conclusion

Questions ?