The Fractal Model

Reflective components for configurable distributed systems

Jean-Bernard Stefani -- INRIA

(joint work with: E. Bruneton, T. Coupaye [Fractal], A. Schmitt [Kell Calculus])

Executive summary

- Programming for large scale, dynamic systems must be component-based programming
  - open systems, constantly evolving, many sources of functionality and service
- Component (run-time entity)
  - = membrane + content
  - = object + reflection
- No pre-defined semantics for component membranes and component bindings
  - component = composition operator
- Components can be shared
  - DAG composition structures (not just trees)
- A "Fractal semantics" can be formally defined
  - abstract co-algebraic one, more concrete operational one

Outline

- Motivations
- Fractal: concepts & principles
- Programming with Fractal: the Julia example
- Kells: co-algebraic foundations for Fractal
- Kell calculus: operational semantics for Fractal
- Perspectives
- Conclusion

Motivations

- Components for computing in the wide: a fact of life
  - plug-ins, xBeans, packages, COM & .Net, etc
- Components: at the crossroad of multiple concerns
  - modularity
  - software architecture
  - unplanned software evolution
  - distribution
  - mobility
  - deployment
  - configuration management

Motivations

- Building dynamically configurable & manageable distributed systems
  - Applications & their software infrastructures (OS & middleware)
  - System software architecture
    - maintenance, reuse, design communication
  - Distributed dynamic configuration
    - distributed deployment, un-planned on-line system/software evolution, adaptive behavior, specialization and optimization
  - Control & management
    - instrumentation, monitoring & controlling behavior
- Limitations in current component programming models & ADLs
  - limited support for extension and adaptation
  - fixed forms of composition
  - fixed forms of introspection & intercession (fixed MOPs)

Fractal

- A component model
  - for building dynamically reconfigurable distributed systems
  - programming language-independent
  - with lightweight implementations (C, C++, Java)
- Used in particular for building distributed systems infrastructures
  - operating systems
  - middleware (application servers, grids, etc)

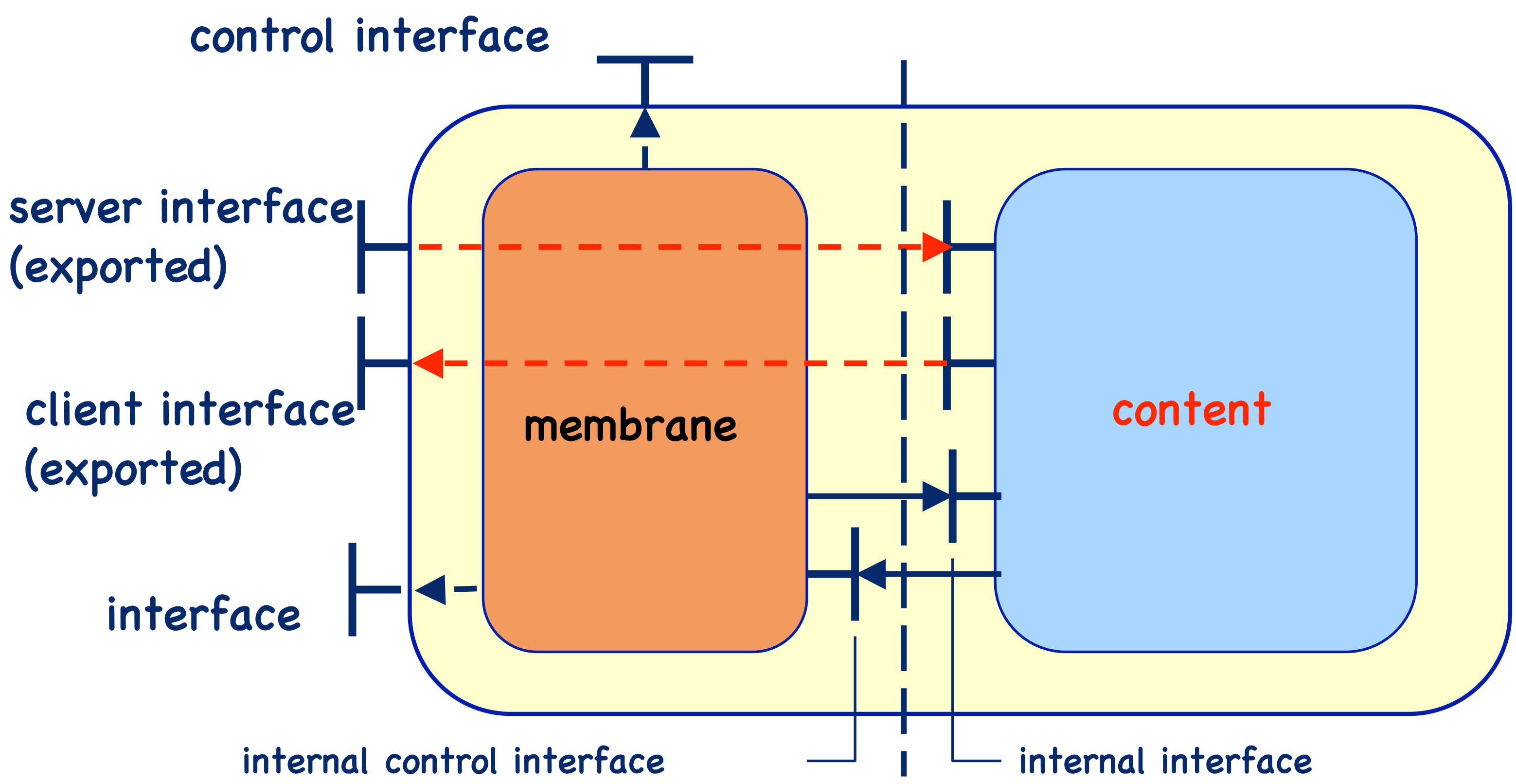
Fractal: « classical » concepts

- *Components* are runtime entities.
  - Not only design time or load time.
- *Interfaces* are the only access points to components.
  - Interfaces emits and receives operation invocations.
- *Bindings* can be primitive (in the same adress space) or composite.
  - In the latter case, they are represented as components and bindings.
  - No fixed semantics for bindings.

Fractal: more original concepts

- A component comprises a ***membrane*** and a ***content***
  - A membrane is made of ***controllers***.
    - It can export control interfaces for some of these controllers.
    - The membrane exercises an arbitrary control over its content.
    - Components can export arbitrary details of their implementation.
    - No fixed meta-object protocol for component introspection & intercession
  - A content is made of other components.
  - A component ***has*** state.
- Components can be ***shared*** by multiple enclosing components.
  - Shared components are crucial for modeling software architectures with resources.

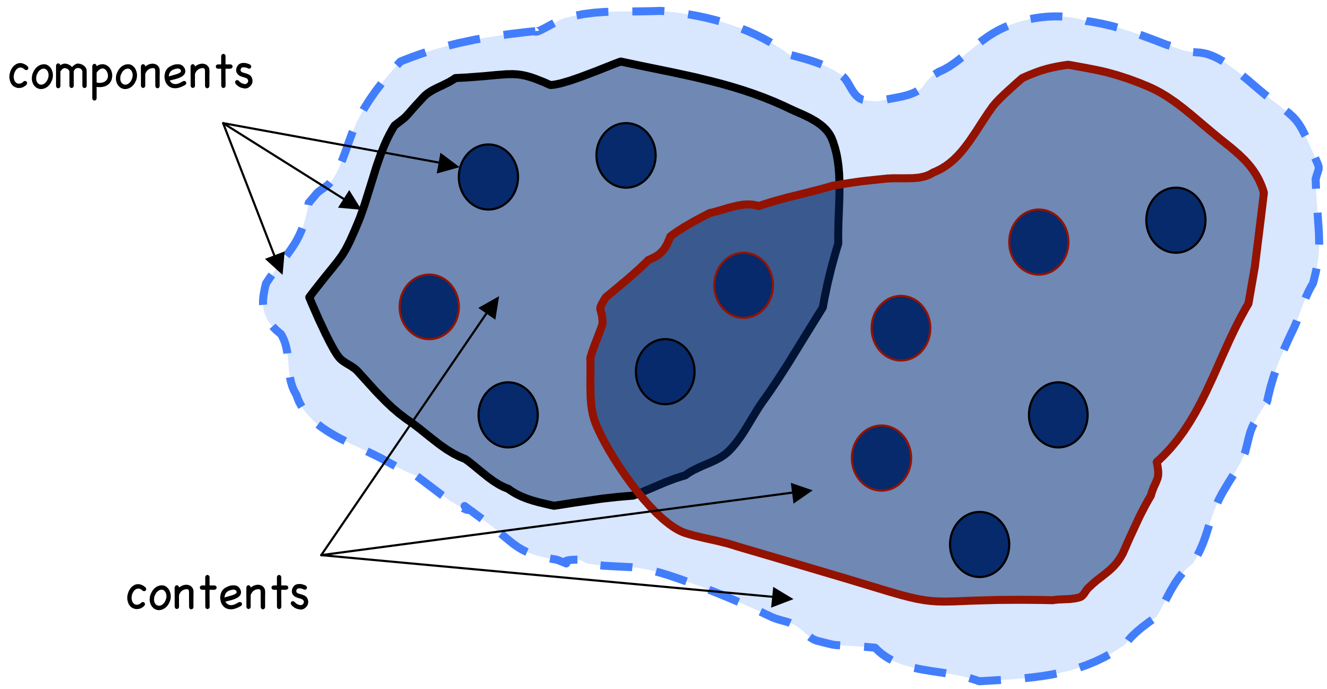
A Fractal Component



Fractal: concepts

- Structure of a component
  - interfaces: named access points (can be “client” or “server”)
  - membrane: set of controllers
  - controllers exercize arbitrary forms of control on the content of a component
  - controllers = meta-objects, meta-groups, advices
  - content: set of components
  - contents may overlap: sharing
- Opaque membrane = no visible control: plain objects
  - dealing with legacy object-based systems

Component sharing



Fractal: useful controllers

- Minimal introspection:
  - Component interface
  - Interface interface
    - cf COM, IUnknown
- Component introspection (I)
  - Content controller
    - to add/remove sub-components
  - Attribute controller
    - to set/get component attributes

Fractal: useful controllers

- Component introspection (II)
  - Binding controller
    - to set up/remove communication paths to/from component
    - a "binding" between components:
      - a component
      - can have arbitrary communication semantics
    - connecting components via a binding involves:
      - creating a binding (component)
      - using binding controllers on components to bind to set up 'primitive bindings' (e.g. language references) with binding (component)
  - Lifecycle controller
    - to start/stop a component

Fractal: additional elements

- Instantiation
  - Factories
    - esp. binding factories
  - Templates: “homomorphic” factories
  - Bootstrap: “well-known” generic factory
- Simple type system
  - Interface
  - Component

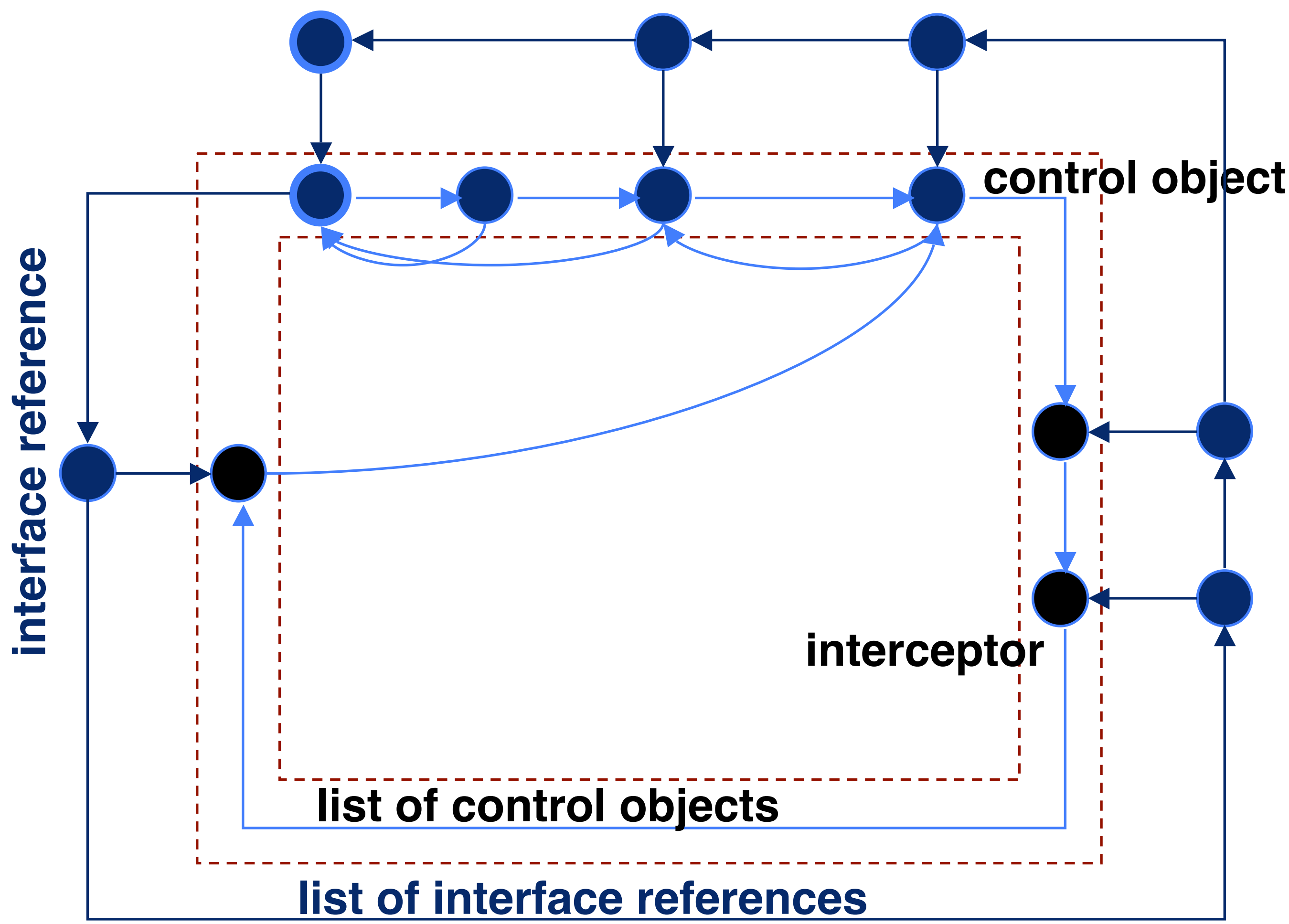
Supporting the Fractal model

- General component structure
  - membrane = set of controllers
  - content = set of components
- No pre-determined control => support must facilitate the definition of membranes
  - library of controllers
    - default ones from Fractal specification
    - interceptors
  - ability to combine controllers
    - e.g. using mixins, components

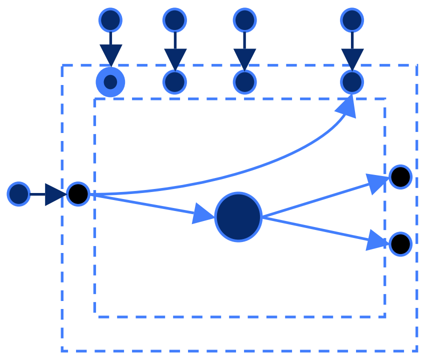
Supporting the Fractal model: Julia

- Supporting Fractal in Java
  - primitive components defined by Java classes
  - primitive bindings are Java references
  - controllers are Java objects
  - controller (mixin) classes can be combined at load-time using a byte-code generator

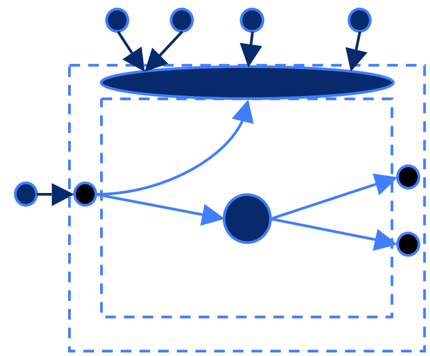
Julia: component structure



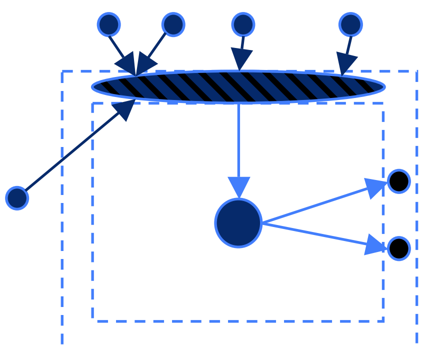
Julia: component structure



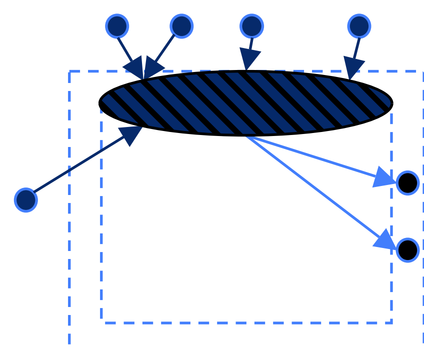
no
optimization



merge of
control
objects



merge of
control objects
& interceptors



merge of
control objects
interceptors,
& content

Fractal: Sample uses

- Operating system kernels
  - Think (FTR&D & INRIA Sardes)
- Asynchronous middleware & communication subsystems
  - DREAM (INRIA Sardes)
- Transaction management
  - GOTM, Jironde (LIFL-INRIA Jacquard, INRIA Sardes)
- Persistency services
  - Speedo, Perseus (FTR&D, LSR)
- Software architecture for Grid applications
  - Proactive (INRIA Oasis)
- Self-adaptive structures
  - (EMN-INRIA Obasco)

Fractal foundations: Kells

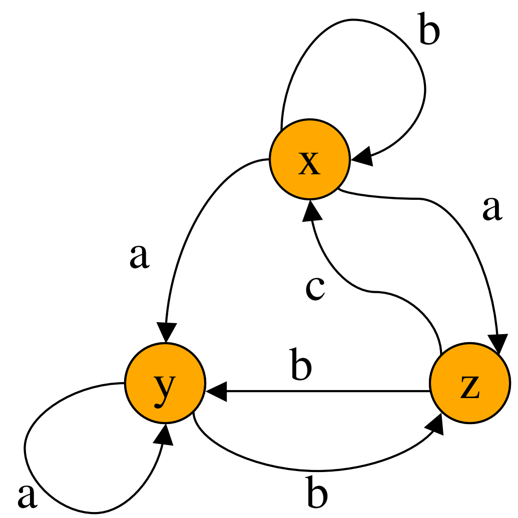
- A kell interacts with its environment through signals
  - signal : [m_1: v_1, …, m_k : v_k]
  - m : label, v : argument
  - arguments can be names (e.g. labels), values and kells
- The behavior of a kell is a collection of possible transitions
  - [content: $M_f(C)$, input: $M_f(S)$, output: $M_f(S)$, residue: $M_f(C)$]
  - content : finite multiset of kells
  - input : finite multiset of signals
  - output : finite multiset of signals
  - residue : resulting configuration (finite multiset of kells)
  - NB: ‘the membrane is the kell’

Fractal foundations: Kells

- A co-algebraic definition of kells
  - characterize kells in a syntax-free manner
  - use hypersets to get final models with a straightforward interpretation
  - hypersets = non-well-founded sets (cf. Aczel, Barwise & Moss)
    - a system of equations is a tuple (X,A,e), where X and A are 2 disjoint sets and e : X ->P(X U A)
    - AFA (Anti-Foundation Axiom): every system of equations (X,A,e) has a unique solution s

Hypersets

- Examples
  - streams : X -> A x X
    - x = < a, y >   y = < b, x>
    - x = abab...   y = baba...
  - automata : X -> P(A x X)
    - x = {<a, y>, <a, z>, <b, x>}
    - y = {<a,y>, <b,z>}
    - z = {<c,x>, <b,y>}



Coalgebras

- Coalgebra
  - An operator G on hypersets is monotone if for all a,b:

    $$a \subset b \Rightarrow G(a) \subset G(b)$$

  - A G-coalgebra is a pair <X,**e**> where X is a set, and **e** is a function

    $$\mathbf{e} : X \rightarrow G(X)$$

- Final coalgebra theorem

  Let G be a monotone operator. Then:

  - G has a greatest fixed point G*,
  - and every G-coalgebra has a unique solution in G*

Kells: formal definition

- Operator G (on hypersets)
  - $G(X) = P(M_f(X) \times M_f(S) \times M_f(S) \times M_f(X))$
  - $S = \cup_{k \in N} (L \times D)^k$
  - D = L + V + X (names + values + kells)
  - P: powerset M_f: finite multisets
- A kell c is the unique solution of a pointed G-coalgebra, <X, **e**, x>
  - <X, **e**> is a G-coalgebra
  - x is an element of X
  - e is a set of (hyperset) equations: **e**: X -> G(X)
  - the solution of <X, **e**, x> is s(x), where s is the solution of <X, **e**>

Example kells

- Simple objects
  - empty content
  - signal arguments : names and values only
- Higher-order objects
  - empty content
- Components with interfaces
  - named access points = receiving signals with target name argument
- Meta-objects, meta-groups
  - M[c], M[a_1, ... a_n]
  - M intercepts, introspects, etc.

Fraktal: Fractal & the Kell calculus

- Kell calculus
  - higher-order π + hierarchical localities + passivation
  - a family of process calculi, parameterized by input patterns (μ)
  - common syntax

P,Q ::= stop	-- inaction
| x	-- process variable
| new a in P	-- restriction
| (μ => P)	-- input
| a<P>.Q	-- output
| (P | Q)	-- parallel composition
| a[P].Q	-- locality or *kell* (strong form)
| a{P}.Q	-- locality (weak form)

Fraktal: Local programming

- Messages: $a\langle\, l_1\langle v_1\rangle \mid \ldots \mid l_n\langle v_n\rangle \,\rangle$
  - a : channel (or interface, or port) name on which messages are sent and received
  - l : parameter name
  - v : parameter value
  - v can be a name, or a program (including another message)
  - Convention: $a\langle\, v_1; \ldots ; v_n \,\rangle = a\langle\, 1\langle v_1\rangle \mid \ldots \mid n\langle v_n\rangle \,\rangle$
- Triggers: (μ => P)
  - μ : input pattern; specifies messages to receive
  - P : program triggered on receipt of messages matching μ

Fraktal: Local programming

- Standard π-calculus congruence rules apply
- Operational semantics

$$M_1 \mid \ldots \mid M_n \mid (\mu \Rightarrow P) \;\to\; P\{x_i := v_i\}$$

  if $M_1, \ldots, M_n$ match μ

  x_i are formal parameters of pattern μ

  v_i are values extracted by μ from messages M_k

- Note: replication can be encoded (standard)
  - $(\mu \Longrightarrow P) = \text{new } t \text{ in } t\langle Y_{\mu,P,t}\rangle \mid Y_{\mu,P,t}$
  - $Y_{\mu,P,t} = (t\langle y\rangle \mid \mu \Rightarrow P \mid t\langle y\rangle \mid y)$

Fraktal: Components

- As in Fractal, components have a membrane and a content: a[P | Q]
  - a[P | Q] : component named "a", with membrane "P", and content "Q"
  - Q must take the form of a parallel composition of components, i.e. $Q = c_1[..] \mid \ldots \mid c_n[..]$
  - P is an arbitrary program, e.g. P can be a parallel composition of components, or simple local programs
- The construct a[.] provides strong encapsulation
  - new a in a[c[Q]] is a perfect firewall : Q cannot communicate with the environment surrounding a

Fraktal: Components

- Patterns for communication across component boundaries: $a<\ldots>^{up:u}$ and $a<\ldots>_{down:u}$
  - $a<\ldots>^{up:u}$ matches a message of the form $a<\ldots>$ coming from the environment of the current component
  - $a<\ldots>_{down:u}$ matches a message of the form $a<\ldots>$ coming from a subcomponent
- Semantics

$$a<v> \mid c[\ (a<z>^{up:u} \Rightarrow P)\ ] \ \rightarrow\ c[\ P\{u:= c,\ z:= v\}\ ]$$

$$c[\ a<v> \mid Q\ ] \mid (a<z>_{down:c} \Rightarrow P) \ \rightarrow\ c[Q] \mid P\{z := v\}$$

Fraktal: Components

- Patterns for matching on sub-components
  - a[x] : pattern that matches a sub-component named a
  - Example: suspending and resuming a subcomponent "a":

Suspend = (suspend<a> | a[x] => c_a<x>)

Resume = (resume<a> | c_a<x> => a[x])

suspend<a> | resume<a> | a[P] | Suspend | Resume

-> resume<a> | c_a<P> | Resume

-> a[P]

Fraktal: Components

- In a component a[P | Q], the membrane "P" may contain several constituent programs, running in parallel
- This is exactly as in Fractal, where a component may have several controllers and interceptors
- Note that the asymmetry between membrane "P" and content "Q" is present due to the constrained form of "Q"

Fraktal: Components

- Programming Fractal-like controllers and interceptors
  - interceptors: routing processes in membranes
  - content controller: adding and removing subcomponents
    - the content Q of component a[P | Q] is supposed to be composed of several components, i.e. Q = c_1[..] | ... | c_n[..].
    - P can maintain a list <c_1,...,c_n> of its subcomponents (e.g. as a message cons<c1; cons<...; cons<cn; nil>...>>)
    - the content controller CC in P (i.e. P = CC | T, for some T), can be written

CC = Add | Remove

Add = (add<w;x>$^{up:y}$ ==> x | addToList<w>)

Remove = (rm<w> | w[y] ==> rmFromList<w>)

Fraktal: Components

- Programming Fractal-like controllers (bis)
  - life-cycle : as in Fractal, allow for the suspension and resumption of sub-components
    - cf previous slide on suspension and resumption of sub-components
    - more sophisticated controls of life-cycle are possible
  - binding controller : as in Fractal, put in place a local binding with an external component (typically a binding component)
    - assume the membrane P in component a[P | Q] maintains a list of client interfaces
    - a binding controller BC in P can be written

$$\mathrm{BC} = (\mathrm{bindL}<a,w,x>^{up:a} \mid \mathrm{isClientItf}<w,t> ==> \mathrm{Bc}(w,x,t))$$

$$\mathrm{Bc}(w,x,t) = (w<z>^{down:t} ==> x<z>)$$

Fraktal: Components

- Binding factories and bindings between components
  - Assume two components a[..] and e[..]
    - Component a has a client interface of name c (i.e. a emits on channel c)
    - Component e has a server interface of name s (i.e. e receives on channel s)
  - A binding factory BF for creating bindings between a and e can be written as follows

$$BF = (\text{bindBF}<a,c,s> \Longrightarrow \text{new } s' \text{ in } B(c,s,s') \mid \text{bindL}<a,c,s'>)$$

$$B(c,s,s') = (s'<x>^{\text{down:}a} \Longrightarrow s<x>)$$

  - BF creates a new binding between c and s

Fraktal: Components

a ((c<z>$^{down:t}$ ==> s'<z>) |)   client

s'<x>$^{down:a}$ ==> s<x>   binding component

e ((s<z>$^{up:b}$ ==> ...) |)   server

Perspectives

- Bisimulation semantics for Fraktal
- Type systems for Fraktal
  - e.g. adapting Hennessy & Yoshida process types
- Dealing with sharing
  - early results obtained with D. Hirschkoff, T. Hirschowitz, D. Pous [GPCE 05]
- Dealing with failures & recoverable actions
  - failure detectors, non-fail-stop models
  - combining micro(nano) reboot and transactions
- Fraktal as a basis for a type-safe, dynamic ADL
  - also a primitive workflow language with reconfiguration capabilities

Conclusion

- Cf. executive summary + perspectives
- Not mentioned
  - extensible ADL
  - code packages as components
  - dynamic code evolution in Fractal/Java
  - towards Fractal v3:
    - combining Fractal & AOP, dynamic ADL, controller libraries, etc
- Links:
  - Web site: http://fractal.objectweb.org
  - mailing list: fractal@objectweb.org