Elements of a Dynamic ADL

Jean-Bernard Stefani

INRIA Rhône-Alpes
Outline

1 Motivation

2 Requirements for a dynamic ADL

3 A toy dynamic ADL : Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
1. Motivation

2. Requirements for a dynamic ADL

3. A toy dynamic ADL: Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
ADL as scripting and workflow language for reconfiguration:
- Expressing dynamic reconfiguration actions
- Capturing control flow as software architecture
- Specifying component behavior
1. Motivation

2. Requirements for a dynamic ADL

3. A toy dynamic ADL: Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
Requirements for a dynamic ADL

Support is required for:

- manipulating attributes (component meta-data)
- manipulating component structure
- manipulating component packages
- manipulating persistent and control flow state
- workflow patterns (e.g. join, split, iterations, etc)
- exception handling
- rollback and recovery
- atomicity conditions
- event-condition-action (ECA) rules
Outline

1 Motivation

2 Requirements for a dynamic ADL

3 A toy dynamic ADL: Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
Constructions:

- Kell calculus basis
  - serves also as an introduction to the Kell calculus...
- higher-order language
  - components and packages are identified
- events are named records
  - record fields as attributes, record names as channels
- synchronization through join patterns
  - workflow patterns as higher-order programs
- name ascription and membranes for reconfiguration
  - uniform handling of state, modelling Fractal membranes
1. Motivation

2. Requirements for a dynamic ADL

3. A toy dynamic ADL: Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
Given sets

Definition

$\mathbb{P}$ is the set of programs: $P, Q, \ldots \in \mathbb{P}$.
$\mathbb{N}$ is the set of names: $a, b, \ldots \in \mathbb{N}$.
$\mathbb{V}$ is the set of variables: $x, y, \ldots \in \mathbb{V}$.
$\mathbb{T}$ is the set of primitive values: $t, v, \ldots \in \mathbb{T}$. 
Components

Definition

\[ C ::= \text{stop} \quad \text{null component} \]
\[ | \quad a[P | C] \quad \text{component} \]
\[ | \quad (a : P) \quad \text{named stuff} \]
\[ | \quad *a \quad \text{shared component} \]
\[ | \quad C | C \quad \text{parallel components} \]
Forms of components:

- \( a[P \mid C] \) denotes a component named \( a \), with membrane \( P \), and content \( C \).
- \( C \) is in general a parallel composition (i.e. a bag) of components.
- \( (a : P) \) is really just a named process (see below). An important use for architecture description is as a binding component.
- \( *a \) is really just a reference to a component named \( a \).

Sharing is done via referencing.

In (untyped) Fraktal, a component interface is manifested as a channel name \( a \), i.e. some name on which messages can be sent.

The component grammar provides for a static description of a component structure. Compare with current Fractal ADL.
Programs

Definition

\[ P, Q ::= \text{stop} \]
\[ x \]
\[ M \]
\[ \text{new } x \text{ in } P \]
\[ P \mid Q \]
\[ (\text{on } J \text{ do } P) \]
\[ \text{case } P \text{ of } J \text{ then } Q \text{ else } R \text{ end} \]
\[ C \]

null process
process variable
message
name creation
parallel composition
reaction rule
case branch
component
Messages and values

Definition

Messages:

\[ M ::= a\langle n_1 : V_1 \mid \ldots \mid n_k : V_k \rangle \]
\[ \quad \mid M \mid M \]

simple message
compound message

Values:

\[ V ::= a \]
\[ \quad \mid v \]
\[ \quad \mid P \]

name
primitive value
program
A primitive message $m \triangleq a\langle n_1 : V_1 \mid \ldots \mid n_k : V_k \rangle$ is just an event (record) $\langle n_1 : V_1 \mid \ldots \mid n_k : V_k \rangle$ sent on a channel (or topic) $a$.

A compound message (event) $M$ is a conjunction of primitive messages (events) : $M \triangleq m_1 \mid \ldots \mid m_k$.

A message can double as a memory cell :

$a\langle V \rangle$ is a memory cell named $a$, holding the value $V$.

Notation : $a\langle V_1; \ldots; V_k \rangle$ stands for $a\langle 1 : V_1 \mid \ldots \mid k : V_k \rangle$. 
The three forms of naming stuff in Fraktal, \( a\langle \ldots \rangle \), \( a[\ldots] \), and \( (a : \ldots) \), are really variations on the same theme.

\( a\langle \ldots \rangle \) is in fact \( (a : \langle \ldots \rangle) \), i.e. a named record.

\( a[\ldots] \) is in fact \( (a : [\ldots]) \), i.e. a named membrane.

One can build complex names for stuff:

\[
(a_1 : (a_2 : \ldots (a_k : V)))
\]

Can be read:

\[
(a_1.a_2.\ldots.a_k : V)
\]
Reaction patterns: shallow introspect

Definition

\[ J ::= \ ? \quad \text{any pattern} \]
\[ \quad a\langle \xi \rangle \quad \text{message value pattern} \]
\[ \quad (a : \xi) \quad \text{named value pattern} \]
\[ \quad a[\xi] \quad \text{component value pattern} \]
\[ \quad *a \quad \text{reference pattern} \]
\[ \quad a\langle J \rangle \quad \text{message pattern} \]
\[ \quad (a : J) \quad \text{named process pattern} \]
\[ \quad a[J] \quad \text{component pattern} \]
\[ \quad J \mid J \quad \text{join pattern} \]

\[ \xi ::= a \mid x \quad \text{match elements} \]
Definition

\[ J ::= \ ? \quad \text{any pattern} \]
\[ \quad \zeta \langle \xi \rangle \quad \text{message value pattern} \]
\[ \quad (\zeta : \xi) \quad \text{named value pattern} \]
\[ \quad \zeta [\xi] \quad \text{component value pattern} \]
\[ \quad *a \quad \text{reference pattern} \]
\[ \quad \zeta \langle J \rangle \quad \text{message pattern} \]
\[ \quad (\zeta : J) \quad \text{named process pattern} \]
\[ \quad \zeta [J] \quad \text{component pattern} \]
\[ \quad J \mid J \quad \text{join pattern} \]

\[ \zeta, \xi ::= a \mid x \quad \text{match elements} \]
Examples of pattern matching

- $a\langle n : x | ? \rangle$ matches $a\langle n : V \rangle$, and yields the substitution $x := V, ? := \text{stop}$.

- $a\langle n : x | ? \rangle$ matches $a\langle n : V | m : P | m' : Q \rangle$, and yields $x := V, ? := (m : P | m' : Q)$.

- $a\langle n : x | m : y \rangle$ matches $a\langle n : V | m : W \rangle$ and yields $x := V, y := W$.

- $a\langle x \rangle | e[c : y]$ matches $a\langle P \rangle | e[c : Q]$ and yields the substitution $x := P, y := Q$. 
Outline

1. Motivation

2. Requirements for a dynamic ADL

3. A toy dynamic ADL: Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
Equivalence laws

- is associative, commutative, has \texttt{stop} as neutral element.

The semantics of programs is given by a reduction relation $\rightarrow$, which specifies the evolution of a program:

\[ P \rightarrow Q \text{ means "in one atomic step, } P \text{ can evolve into } Q" \]

The reduction relation is defined by induction rules of the form:

\[
\begin{array}{c}
\text{Premise} \\
\hline
\text{Conclusion} \\
\end{array} 
\text{[RULE.NAME]}
\]

meaning: if Premise is true, then infer Conclusion.

It is always possible to apply a reduction step in the following contexts:

\[
E ::= \cdot \mid E \mid P \mid a[E] \mid (a : E)
\]
Local rules

Definition

\[
\frac{a \text{ fresh}}{\text{new } x \text{ in } P \rightarrow P\{ a/x \}} \quad [\text{NEW}]
\]

\[
\frac{\text{match}(J, Q) = \theta}{(\text{on } J \text{ do } P) \mid Q \rightarrow P\theta} \quad [\text{LOCAL}]
\]

\[
\frac{\text{match}(J, P) = \theta}{\text{case } P \text{ of } J \text{ then } Q \text{ else } R \text{ end} \rightarrow Q\theta} \quad [\text{CASE.T}]
\]

\[
\frac{\neg \text{match}(J, P)}{\text{case } P \text{ of } J \text{ then } Q \text{ else } R \text{ end} \rightarrow R} \quad [\text{CASE.F}]
\]
Rules for crossing boundaries

Definition

\[
\begin{align*}
\text{match}(J, Q | M) &= \theta \\
\text{match}(J, Q | M) &= \theta
\end{align*}
\]

\[
\begin{align*}
a[(\text{on } J \text{ do } P) | Q | S] | M &\rightarrow a[P\theta | S] & \text{[IN]} \\
(\text{on } J \text{ do } P) | Q | a[M | S] &\rightarrow P\theta | a[S] & \text{[OUT]}
\end{align*}
\]
Rules for named programs

**Definition**

\[
(on \ J \ do \ P) \mid \prod_{i=1}^{n} Q_i \rightarrow P^\theta \\
E\{ (on \ J \ do \ P), Q_1, \ldots, Q_n \} \rightarrow E\{ P^\theta, \text{stop}, \ldots, \text{stop} \} \quad [\text{NAME}]
\]

\[
c[ (on \ J \ do \ P) \mid Q \mid \prod_{i=1}^{n} Q_i ] \mid \prod_{j=1}^{m} P_j \rightarrow P^\theta \\
A \rightarrow B \quad [\text{NAME.K}]
\]

where

\[
A = E_2\{ c[ E_1\{ (on \ J \ do \ P), Q, Q_1, \ldots, Q_n \ }, P_1, \ldots, P_m \} \\
B = E_2\{ c[ E_1\{ P^\theta, Q, \text{stop}, \ldots, \text{stop} \ }, \text{stop}, \ldots, \text{stop} \} 
\]

and \(E, E_1, E_2\) are a named contexts.
A named context $E$ is a multihole context of the form:

$$E ::= i | E | P | (a : E)$$
Rule for shared components

Definition

\[
\begin{align*}
\text{E} \{ a[ P \mid (b : Q)] \} &\rightarrow \text{E} \{ a[ P' \mid (b : Q')] \} \\
c[ E \{ a[ P \mid *b] \} \mid (b : Q)] &\rightarrow c[ E \{ a[ P' \mid *b] \} \mid (b : Q')] \\
\end{align*}
\]

[S\text{HD}]

where \( E \) is a context of the form:

\[
E ::= \cdot \mid E \mid T \mid (a : E) \mid a[E]
\]

where \( T \neq b[R] \mid S, (b : R) \mid S \)
Outline

1 Motivation

2 Requirements for a dynamic ADL

3 A toy dynamic ADL: Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
Support for ECA expressions

- Reaction rules in Fraktal follow the ECA format
- Pattern matching provides for complex condition expressions
- Join patterns constitute expressive event language (conjunction, disjunction)
- More sophisticated event / conditions can be programmed (i.e. behavioral predicate checking through transducers)
Manipulating attributes

- An attribute is a private cell, together with getter and setter functions.

- Let $A \triangleq a[ P \mid C ]$ be a component. An attribute $a$ of $A$ can be defined as

$$P \triangleq \ldots \mid \text{Getter}(a) \mid \text{Setter}(a) \mid c\langle a; V \rangle$$

$$\text{Getter}(a) \triangleq (\text{on get}\langle a; r \rangle \mid c\langle a; x \rangle \text{do } r\langle x \rangle \mid c\langle a; x \rangle)$$

$$\text{Setter}(a) \triangleq (\text{on set}\langle a; x; r \rangle \mid c\langle a; y \rangle \text{do } r\langle x \rangle \mid c\langle a; x \rangle)$$
Patterns of the form $a\langle J \rangle$, $(a : J)$ and $a[J]$ can introspect a structure (message or component) at an arbitrary deep level.

Reaction rules and case branches allow to modify a component structure in arbitrary ways.

One can extend the pattern language to a navigation language à la Fpath [P.C David, PhD ‘05] (similar level of instrospection, different queries), and introduce a transform operator à la CDuce.

This turns Fraktal into a concurrent version of Safran/Fscript.
Manipulating component packages

- A package in Fraktal is a component that is suspended in a cell!
- Fraktal is a higher-order language of (component) prototypes.
- One could think of a ‘class-based’ version:
  - Primitive classes and membrane classes would be records of reactions and cells.
  - A general component class would be a component with classes subcomponents.
  - The dependency of components to component classes would be manifested by the sharing of the component class as a subcomponent of each of its instances (a mere reference).
State in a component can be accessed if named: via messages, named stuff, or components.

In Fraktal, control flow state and persistent state is not strictly distinguished. Control flow state is a mixture of reaction rules, case branches and messages or named stuff they use.

If all reaction rules and case branches are named, all the control flow state is accessible through **deep introspect patterns**, provided certain naming conventions are adhered to.

- Example naming convention: a reaction rule or case branch gets a unique name, created by the concatenation of a unique name and a name that uniquely distinguishes the rule or branch in the program text.

- Control flow state can be accessed with just **shallow introspect patterns** provided each named process maintains a known name log.
Workflow patterns

See work on Biztalk semantics and workflow patterns in the \( \pi \)-calculus [Puhlmann, BPM 05].

Higher-order in Fraktal allows to build workflow structures dynamically.

Workflow = some Fraktal program
- activity = program (typically, a component)
- workflow pattern = higher-order program

For instance
- AND synchronization on \( n \) activity terminations \( t_i \) for starting activity
  \( S : \)
  \[
  AND\langle T_1, \ldots, T_n \rangle = \prod_{i=1}^{n} T_i \mid (on \prod_{i=1}^{n} t_i \text{ do } S)
  \]
- OR synchronization on \( n \) activity terminations \( t_i \) for starting activity
  \( S : \)
  \[
  OR\langle T_1, \ldots, T_n \rangle = \prod_{i=1}^{n} T_i \mid \text{new } k \text{ in } k \mid \prod_{i=1}^{n} (on \ t_i \mid k \text{ do } S)
  \]
Exercises!

- Exception handling can be programmed as in Mozart [Van Roy, CTM MIT Press 04], and in Erlang [Armstrong, PhD 03]. Exercize: program Erlang failure detection channels and trees of failure handlers, in Fraktal.

- Rollback and recovery can be programmed by mimicking reversible processes as in [Danos, CONCUR 04].

- Transactions can be programmed as in [Bruni, CONCUR 02].

- Rollback, recovery, and transactions cry out for abstractions of their own...

The transactional solution satisfies one of our criteria for adding a concept to the computation model, namely that programs in the extended model are simpler. But what exactly is the concept to be added? This is still an open research subject. In our view it is a very important one. [P. Van Roy, CTM pp.602]
Outline

1. Motivation

2. Requirements for a dynamic ADL

3. A toy dynamic ADL: Fraktal
   - Syntax
   - Semantics
   - Meeting the requirements
   - Supporting Fraktal
Do not use Java concurrency constructs in component code...

- Treat each reaction as a chord in [Polyphonic C#, TOPLAS 03].
  - Methods in join patterns annotated with synch and asynch property.
  - At most one synch method per join pattern.

- Treat each reaction as an activity in the Dream framework [Quema, PhD ?]. Mapping to Java threads is left to explicit schedulers.

- Allow long-running reactions, provided they obey the atomicity property of reactions (no visible effect apart from input and output of messages/components).
  - Transient reaction state is not visible in Fraktal.
A possible implementation strategy for Fraktal/Java

- Provide support for 3 alternatives: running to completion (short-lived activities), rollback (long-lived ones), abort on consistent state (long-lived ones).

  - **Running to completion and abort on consistent state**
    - nothing to do
    - rely on programmer’s adherence to guidelines
    - risk: inconsistent persistent state as visible in Fraktal

  - **Abort on consistent state** requires programming guidelines for complex data structures
    - complex data structure in reaction workspace locked in Fraktal cells should be closed (no escape reference).
    - concurrent access to those should be spread across several Fraktal cells.
    - can be supplemented by persistent consistency flag.

- Overhead in normal execution for rollback can be reduced by explicit reaction log made persistent in Fraktal cell.
Reconfiguration programs can be arbitrarily complex
Support should comprise of a dynamic ADL (doubling as a higher-order workflow language)

Fraktal provides toy example of dynamic Fractal ADL
Fraktal/Java implementation strategy to be explored
Other programming languages worth investigating...