Elements of a Dynamic ADL

Jean-Bernard Stefani

INRIA Rhône-Alpes

Outline

Outline

Motivation

ADL as scripting and workflow language for reconfiguration :

- Expressing dynamic reconfiguration actions
- Capturing control flow as software architecture
- Specifying component behavior

Outline

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

A toy dynamic ADL : 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

Outline

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

$C ::= \mathbf{stop}$		*null component*
	$\mid \; a[P \mid C]$	*component*
	$\mid \; (a : P)$	*named stuff*
	$\mid \; *a$	*shared component*
	$\mid \; C \mid C$	*parallel components*

Comments on 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

$$
\begin{array}{lr}
P, Q ::= \textbf{stop} & \textit{null process} \\
\quad \mid \; x & \textit{process variable} \\
\quad \mid \; M & \textit{message} \\
\quad \mid \; \textbf{new}\, x\, \textbf{in}\, P & \textit{name creation} \\
\quad \mid \; P \mid Q & \textit{parallel composition} \\
\quad \mid \; (\textbf{on}\, J\, \textbf{do}\, P) & \textit{reaction rule} \\
\quad \mid \; \textbf{case}\, P\, \textbf{of}\, J\, \textbf{then}\, Q\, \textbf{else}\, R\, \textbf{end} & \textit{case branch} \\
\quad \mid \; C & \textit{component}
\end{array}
$$

Messages and values

Definition

Messages :

$$M ::= a\langle n_1 : V_1 \mid \ldots \mid n_k : V_k \rangle \quad \textit{simple message}$$
$$\mid \quad M \mid M \quad \textit{compound message}$$

Values :

$$V ::= a \quad \textit{name}$$
$$\mid \quad v \quad \textit{primitive value}$$
$$\mid \quad P \quad \textit{program}$$

Comments on messages

- 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 $a\langle\, 1 : V_1 \mid \ldots \mid k : V_k \,\rangle$.

Comments on named stuff

- 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

$J ::= \ ?$		*any pattern*
	$\mid \ a\langle \xi \rangle$	*message value pattern*
	$\mid \ (a : \xi)$	*named value pattern*
	$\mid \ a[\xi]$	*component value pattern*
	$\mid \ *a$	*reference pattern*
	$\mid \ a\langle J \rangle$	*message pattern*
	$\mid \ (a : J)$	*named process pattern*
	$\mid \ a[J]$	*component pattern*
	$\mid \ J \mid J$	*join pattern*
$\xi ::= a$	$\mid \ x$	*match elements*

Reaction patterns : deep introspect

Definition

$J ::=$	$?$	*any pattern*
	$\mid \zeta\langle \xi \rangle$	*message value pattern*
	$\mid (\zeta : \xi)$	*named value pattern*
	$\mid \zeta[\xi]$	*component value pattern*
	$\mid *a$	*reference pattern*
	$\mid \zeta\langle J \rangle$	*message pattern*
	$\mid (\zeta : J)$	*named process pattern*
	$\mid \zeta[J]$	*component pattern*
	$\mid J \mid J$	*join pattern*
$\zeta, \xi ::= a$	$\mid x$	*match elements*

Examples of pattern matching

- $a\langle n : x \mid ?\rangle$ matches $a\langle n : V\rangle$, and yields the substitution $x := V, ? := \textbf{stop}$.
- $a\langle n : x \mid ?\rangle$ matches $a\langle n : V \mid m : P \mid m' : Q\rangle$, and yields $x := V, ? := (m : P \mid m' : Q)$.
- $a\langle n : x \mid m : y\rangle$ matches $a\langle n : V \mid m : W\rangle$ and yields $x := V, y := W$.
- $a\langle x\rangle \mid e[c : y]$ matches $a\langle P\rangle \mid e[c : Q]$ and yields the substitution $x := P, y := Q$.

Outline

Equivalence laws

- $|$ is associative, commutative, has **stop** as neutral element.
- The semantics of programs is given by a reduction relation $\rightarrow$, which specifies the evolution of a program :

  $P \rightarrow Q$ means "in one atomic step, P can evolve into Q"
- The reduction relation is defined by induction rules of the form :

$$\frac{\texttt{Premise}}{\texttt{Conclusion}}\ [\textsc{Rule.name}]$$

  meaning : if Premise is true, then infer Conclusion.
- It is always possible to apply a reduction step in the following contexts :

$$\mathbf{E} ::= \cdot \quad | \quad \mathbf{E} \,|\, P \quad | \quad a[\mathbf{E}] \quad | \quad (a : \mathbf{E})$$

Local rules

Definition

$$\frac{a \text{ fresh}}{\textbf{new}\, x\, \textbf{in}\, P \rightarrow P\{{}^{a}/_{x}\}}\ [\textsc{New}] \qquad \frac{\texttt{match}(J, Q) = \theta}{(\textbf{on}\, J\, \textbf{do}\, P) \mid Q \rightarrow P\theta}\ [\textsc{Local}]$$

$$\frac{\texttt{match}(J, P) = \theta}{\textbf{case}\, P\, \textbf{of}\, J\, \textbf{then}\, Q\, \textbf{else}\, R\, \textbf{end} \rightarrow Q\theta}\ [\textsc{Case.T}]$$

$$\frac{\neg\texttt{match}(J, P)}{\textbf{case}\, P\, \textbf{of}\, J\, \textbf{then}\, Q\, \textbf{else}\, R\, \textbf{end} \rightarrow R}\ [\textsc{Case.F}]$$

Rules for crossing boundaries

Definition

$$\frac{\texttt{match}(J, Q \mid M) = \theta}{a[\,(\textbf{on}\, J\, \textbf{do}\, P) \mid Q \mid S\,] \mid M \rightarrow a[\,P\theta \mid S\,]}\ [\textsc{In}]$$

$$\frac{\texttt{match}(J, Q \mid M) = \theta}{(\textbf{on}\, J\, \textbf{do}\, P) \mid Q \mid a[\,M \mid S\,] \rightarrow P\theta \mid a[\,S\,]}\ [\textsc{Out}]$$

Rules for named programs

Definition

$$\frac{(\textbf{on}\ J\ \textbf{do}\ P) \mid \prod_{i=1}^{n} Q_i \rightarrow P\theta}{\mathbf{E}\{\,(\textbf{on}\ J\ \textbf{do}\ P), Q_1, \ldots, Q_n\,\} \rightarrow \mathbf{E}\{\,P\theta, \textbf{stop}, \ldots, \textbf{stop}\,\}}\ [\textsc{Name}]$$

$$\frac{c[\,(\textbf{on}\ J\ \textbf{do}\ P) \mid Q \mid \prod_{i=1}^{n} Q_i\,] \mid \prod_{j=1}^{m} P_j \rightarrow P\theta}{A \rightarrow B}\ [\textsc{Name.K}]$$

where

$$A = \mathbf{E}_2\{\,c[\,\mathbf{E}_1\{\,(\textbf{on}\ J\ \textbf{do}\ P), Q, Q_1, \ldots, Q_n\,\}\,], P_1, \ldots, P_m\,\}$$
$$B = \mathbf{E}_2\{\,c[\,\mathbf{E}_1\{\,P\theta, Q, \textbf{stop}, \ldots, \textbf{stop}\,\}\,], \textbf{stop}, \ldots, \textbf{stop}\,\}$$

and $\mathbf{E}, \mathbf{E}_1, \mathbf{E}_2$ are a named contexts.

Named contexts

Definition

A named context **E** is a multihole context of the form :

$$\mathbf{E} ::= \cdot_i \quad | \quad \mathbf{E} \,|\, P \quad | \quad (a : \mathbf{E})$$

Rule for shared components

Definition

$$\frac{\mathbf{E}\{\, a[\, P \mid (b : Q)\,]\,\} \rightarrow \mathbf{E}\{\, a[\, P' \mid (b : Q')\,]\,\}}{c[\,\mathbf{E}\{\, a[\, P \mid *b\,]\,\} \mid (b : Q)\,] \rightarrow c[\,\mathbf{E}\{\, a[\, P' \mid *b\,]\,\} \mid (b : Q')\,]}\ [\textsc{Shd}]$$

where $\mathbf{E}$ is a context of the form :

$$\mathbf{E} ::= \cdot \quad | \quad \mathbf{E} \mid T \quad | \quad (a : \mathbf{E}) \quad | \quad a[\,\mathbf{E}\,]$$

where $T \not\equiv b[\,R\,] \mid S, (b : R) \mid S$

Outline

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

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

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

$$\texttt{Setter}(a) \triangleq (\textbf{on}\ \texttt{set}\langle\, a; x; r \,\rangle \mid c\langle\, a; y \,\rangle\ \textbf{do}\ r\langle\, x \,\rangle \mid c\langle\, a; x \,\rangle)$$

Manipulating component structure

- 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).

Manipulating persistent and control state

- 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 π-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 :

$$AND\langle T_1, \ldots, T_n \rangle = \prod_{i=1}^{n} T_i \mid (\textbf{on} \prod_{i=1}^{n} t_i \textbf{ do } S)$$

  - OR synchronization on n activity terminations t_i for starting activity S :

$$OR\langle T_1, \ldots, T_n \rangle = \prod_{i=1}^{n} T_i \mid \textbf{new } k \textbf{ in } k \mid \prod_{i=1}^{n} (\textbf{on } t_i \mid k \textbf{ do } S)$$

Exception handling, rollback, recovery, transactions

Exercizes !

- 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 satifies 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

A possible implementation strategy for Fraktal/Java

- 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.

Summary

- 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...