TOWARDS HIERARCHICAL MANAGEMENT OF AUTONOMIC COMPONENTS: A CASE STUDY

Euromicro PDP 2009, 18-21 Feb, Weimar, Germany

Marco Aldinucci
Computer Science Dept. - University of Torino - ITALY

Marco Danelutto
Computer Science Dept. - University of Pisa - ITALY

Peter Kilpatrick
Computer Science Dept. - Queen’s University Belfast - U.K.
Outline

* A semi-formal framework for autonomic components
  * rigorously defining autonomic cycle
  * rigorously defining managers behaviour

* Behavioural skeletons and component hierarchy
  * decoupling management from business code, coupling behaviour with skeletons/patterns
  * easing autonomic applications design by way of automatic manager generation

* Demo*nstrating all above
Autonomic Components

- CoreGRID Grid Component Model (GCM)
  - recently standardised by ETSI (July 2008)
  - not only grid, but also distributed and multi/may core
- GCM
  - use-provide ports, RPC, events, streams
  - broadcast, multicast, unicast, gather ports
  - hierarchic: components can be nested, derived from Fractal component model, which don’t cover concurrency/parallelism
  - prototypal implementation (GCM/Proactive)
  - autonomic features designed and developed in GridCOMP
  - already used for real world application, see IBM, GridSystem, ATOS, ...
Autonomic Components

Analyse
Is the contract broken? Why?
QoS contract
Plan
Which plan can solve the problem?
Monitor
How is AE behaving?
Adapt
Execute the reconf. protocol

Sensors
Effectors

Wednesday, February 18, 2009
Manager life cycle

Autonomic Components

Analyse
Is the contract broken? Why?

QoS contract

Plan
Which plan can solve the problem?

Monitor
How is AE behaving?

Adapt
Execute the reconf. protocol

Sensors

Effectors
Autonomic Components

Analyse
Is the contract broken? Why?

QoS contract

Plan
Which plan can solve the problem?

Monitor
How is AE behaving?

Adapt
Execute the reconf. protocol

Sensors

Effectors
AC - Components in Insulation

- They are components
  - unit of deployment, legacy code, well-defined dependencies, XML-style assembly, etc.
- Autonomic Components exhibit self-* features
  - self-optimising, self-configuring self-protecting, self-healing
- They can have one or more managers
  - we assumed one, since components can be nested the assumption does not break generality
Component interaction

- Legacy (Cl) no interaction, empty manager, no NF ports
- Passive (Cp) one-way interaction, monitor only capability, read-only NF ports
- Active (Ca) two-ways interactions, monitor and steering capability, read/write NF ports

“less general” components can be nested into “more general” components, but not vice-versa

Cl ⊂ Cp ⊂ Ca
Assembly of Autonomic Components

Management overlay I

Management overlay II
Monitor data is analysed
Management actions are planned

Monitor data (polls/events) can be synthesised (bottom-up)

Steering actions and new contracts can be diffused (top-down)
**Overlay of Managers**

- $C_x = \text{Component } x$
- $C_x', C_x'' = \text{Instances of } C_x$
- $M(C_x) = \text{Manager of } C_x$

**Diagram Notes:**
- Structural relationships
- Functional network
- Management overlay network
- QoS contract (from users)
e.g. \((k_{\text{low}} \leq m_1^1 \leq k_{\text{high}}) \land m_2^1 \leq \theta\)

\[\text{ContractPredicate}(\overline{m}_1, \ldots, \overline{m}_n)\]

Valid

\[t = t + 1\]

Broken contract
goto planning phase
The manager chooses a plan among defined ones

including the empty plan, i.e. better to do nothing

A plan is composed of

1. A reconfiguration protocols (composed of actions)
   - migrate C1 on Platform2; clone C2 and wire it to C1;
   - actions can also consist in communications with other managers

2. Expected benefit and overhead
   - quantified as alteration of monitor variables at some future iteration
   - e.g. increase throughput using more resources $m_i(\bigcirc_{t+k})$
   - $m_0(\bigcirc_{t+3}) = g(m_1(\bigcirc_t), m_2(\bigcirc_t)), m_1(\bigcirc_{t+3}) = f(m_2(\bigcirc_t))$
Which is the better plan?

The one that gives the best expected benefit - cost

- according to a give logic
  - we used first-order logic, but other are viable (e.g. fuzzy)
  - possibly after projecting the n-space of results onto a user-defined goal function

Is there any guarantee that everything will work as expected

No. It is a speculation, but

- It is control loop theory from the mid of last century
- We can reach a good sub-optimum by iterating the process
  - this reduces the forecast window
  - this take in account changing enviroment
ON WHY PROGRAMMING AC IS NEARLY A NIGHTMARE ... AND WHY WE INTRODUCED BEHAVIOURAL SKELETONS
AC idea is basically a vision

- the definition “per se” does not help so much in designing self-management applications

- writing a manager is pretty complex
  - should be decoupled and independent from functional code
  - should preserve semantics of functional code
  - should provide effective management capabilities

- when applied to components it may specialise them too much
  - losing reusability, that is one of key advantages of components
Expressing managers might be complex

User goals are often multi-purpose

- Performance: the app should sustain $x$ transactions per second; the app should complete each transaction in $t$ seconds
- Security: the link between $P_1$ and $P_2$ should be secured with $k$-strong encryption; the DB service is exposed by platform $P_3$
- Fault-tolerance: the parallel server should survive to the failure of $y$ platforms

User wishes are referred to a dynamic world

- ... consider that $x$, $t$, $P_1$, $P_2$, $P_3$, $k$, $y$ can dynamically change as may dynamically change the performance and the state of the running environment
Ideal application management is distributed

* but user wishes (goal/contracts) are atomically expressed
* user would not specify how each part of the (evolving) system contribute to their wishes, and how parts compose w.r.t. goal

The framework previously presented attacks the problem

* gives you a well-defined methodology
  * monitor can be collected bottom-up, steering proceed top-down, management happens finding the minimum common ancestor in a tree of autonomic components
* but does not solve the problems in all cases
  * what happens if the application is composed of two parts user wishes cannot be automatically split in two parts?
**Behavioural Skeletons (BeSke)**

- Represent an evolution of the algorithmic skeleton concept for component management
- Abstract parametric paradigms of component assembly
- Specialised to solve one or more management goals
  - Self-configuration/optimization/healing/protection.
- Carry a semi-formal/formal description and an implementation
  - They are component factories, actually

- Are higher-order components
- Are not exclusive
  - Can be composed with non-skeletal assemblies via standard components connectors
  - Overcome a classic limitation of skeletal systems
Managers Interaction is Well-defined

* Can be formally specified, e.g. using Orc (Cook & Misra)

\[ BSkel(farm(N), contract) \triangleq farm(N) \mid manager(farm(N), contract) \]

\[ farm(N) \triangleq (\mid 1 \leq i \leq N : W_i) \]

\[ W_i \triangleq \]
\[ (\text{if}(b) \gg (W_i.execute(x) > y > \text{out.put}(y) \gg W_i)) \]
\[ \mid \text{if}(\neg b) \gg 0) \]

where \((x, b) :\in\)
\[ (\text{in.get} > y > \text{let}(y, true) \]
\[ \mid \text{Interrupt}_i.get > y > \text{let}(y, false)) \]

\[ \text{adapt}(farm(N), plan) \triangleq \]
\[ (\text{if}(plan = \text{addworker}) \gg \text{let}(y) \gg farm(N + 1) \]

where \((\forall i :: y_i :\in \text{Interrupt}_i.set)\)
1. Choose a schema
e.g. functional replication
ABC API is chosen accordingly

2. Choose an inner component
compliant to BeSke constraints

3. Choose behaviour of ports
e.g. unicast/from_any, scatter/gather

4. Run your application
then trigger adaptations

5. Automatise the process
with a Manager

ABC = Autonomic Behaviour Controller (implements mechanisms)
AM = Autonomic Manager (implements policies)
B/LC = Binding + Lifecycle Controller
Beske families

- Functional Replication
  - Farm/parameter sweep (self-optimization)
  - Stateful Data-Parallel (self-configuring map-reduce)
  - Active/Passive Replication (self-healing)

- Proxy
  - Pipeline (coupled self-protecting proxies)

- Wrappers
  - Facade (self-protection)

- Many others can be borrowed from Design Patterns
Why BeSke are an Advance

* We can associate a standard manager to each BeSke
  * contracts can be predefined, implementation can be automatically generated (by way of a factory)

* BeSke are compositional
  * when nested we can automatically derive the global behaviour of the assembly that is managed in fully distributed way
  * they can be wired in arbitrary graphs
    * in this way the previous property is not always true

* A prototypal implementation exists (GPL)
  * download from my home page http://www.di.unipi.it/~aldinuc
  * managers implemented as JBoss engines, see references
<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Manager Contract</th>
<th>$m_i$</th>
</tr>
</thead>
</table>
| $C_1$     | active pipe    | $K_{\text{low}} \leq T_{\text{self}} \leq K_{\text{high}}$ (user defined) | $K_{\text{low}}, K_{\text{high}}$ constants; $T_{C_2}, T_{C_3}, T_{C_4}$ monitored $T_{\text{self}} = \max\{T_{C_2}, T_{C_3}, T_{C_4}\}$  

<p>| $C_2, 4, 6, 7, 8$ | passive seq    | none             | provide $T_{C_{2,4,6,7,8}}$ via NF port (respectively) |</p>
<table>
<thead>
<tr>
<th>Plan</th>
<th>Expected Cost</th>
<th>Expected Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL(_F_1) move the slower worker (C_w) to a faster platform, if any</td>
<td>cost(\text{stop}(C_w)); deploy(C_w); start(C_w))</td>
<td>decrease service time. (T_{\text{farm}}(\bigotimes t+h) = \delta T_{C_w}(\bigotimes t),\ 0 \leq \delta \leq 1) speed difference between the platforms</td>
</tr>
<tr>
<td>PL(_F_2) increase parallelism degree (allocate (k) new workers)</td>
<td>cost(\text{deploy}(C_{w_j});\text{start}(C_{w_j})) for (j = 1..k) instances</td>
<td>decrease service time. (T_{\text{farm}}(\bigotimes t+h) = \delta T_{\text{farm}}(\bigotimes t)) (\delta = n/(n+k))</td>
</tr>
<tr>
<td>PL(_F_3) decrease parallelism degree (de-allocate (k) workers)</td>
<td>cost(\text{stop}(C_{w_j})) for (j = 1..k) instances</td>
<td>increase service time. (T_{\text{farm}}(\bigotimes t+h) = \delta T_{\text{farm}}(\bigotimes t)) (\delta = (n+k)/n)</td>
</tr>
<tr>
<td>PL(_F_4) raise violation</td>
<td>0 ((\text{negligible}))</td>
<td>none</td>
</tr>
<tr>
<td>PL(_P_1) move stage ((C_s)) with maximum (T) to a faster resource, if any</td>
<td>cost(\text{stop}(C_s);\text{deploy}(C_s);\text{start}(C_s)))</td>
<td>decrease service time. (T_{\text{pipe}}(\bigotimes t+h) = \delta T_{\text{pipe}}(\bigotimes t),\ 0 \leq \delta \leq 1) speed difference between the platforms if (\max{T_{C_s}, T_{\text{pipe}}(C_1, \ldots, C_{s-1}, C_{s+1}, \ldots, C_k)} = T_{C_s}) otherwise (\delta = 1)</td>
</tr>
<tr>
<td>PL(<em>P_2) collapse adjacent stages (C_s, C</em>{s+1})</td>
<td>cost(\text{stop}(C_s);\text{deploy}(C_s);\text{start}(C_s))) for (C_s) and (C_{s+1})</td>
<td>decrease resource usage (n = n - 1). increase service time. (T_{\text{pipe}}(\bigotimes t+h) = \delta + T_{\text{pipe}}(\bigotimes t),\ \delta = 0) iff (T_{C_s} + T_{C_{s+1}} \leq T_{\text{pipe}}(\bigotimes t),\ \delta = T_{C_s} + T_{C_{s+1}} - T_{\text{pipe}}(\bigotimes t)) otherwise</td>
</tr>
<tr>
<td>PL(_P_3) raise violation</td>
<td>0 ((\text{negligible}))</td>
<td>none</td>
</tr>
</tbody>
</table>
DEMO
EXAMPLE: MAMMOGRAPHY

Segmentation
Dicom decoder
Image DB
Dicom decoder

Alice
age...
Slice 0
Slice 1
Slice 2
Slice n

Top Manager
Farm Manager

Segmentation
Segmentation
Segmentation

Background removal
Anisotropic filtering
Thorax Removal & Watershed

Output widget

Movement correction
Classification & Washout

Curves in the 4th dimensional space (X,Y,Time,Enhancement)

IEEE IPDPS, Roma, May 2009
MANDELBROT EXAMPLE (TWO-LEVELS)

demol

app
manager

dataset
generator

farm
manager

mandel
worker

mandel
worker

mandel
worker

screen

↑ get_service_time
↓ change // degree

↑ raise violation
↓ new contract (e.g. Ts<k)

unicast

from_any
We have outlined a framework suitable for modelling hierarchical autonomic management not only for grid: clouds, distributed, multi/many core, ...

We enriched the framework with behavioural skeletons previously existing only “in insulation”

contracts and manager implementation can be automatically generated also in case of composition

We implemented (GCM); we got a ETSI standard

we show with a demo they are effective

has been elected in Jan 2009 “EC ITC project of the month”


