Components, GCM, and Behavioural skeletons

Marco Aldinucci
University of Pisa, Italy
(CoreGRID REP Programme)

M. Danelutto, S. Campa
University of Pisa, Italy

P. Kilpatrick
QUB, UK

N. Tonellotto, P. Dazzi
ISTI-CNR, Italy

November 22th, 2007
QUB, Belfast, UK
OUTLINE

• Prelude
  • Uni. Pisa and the HPC lab.

• Motivation
  • why adaptive and autonomic management
  • why skeletons

• Behavioural Skeletons
  • parametric composite component with management
  • functional and non-functional description
  • families of behavioural skeletons

• GCM implementation
  • preliminary experiments and performances
Computer Science Dept.
- First in Italy (estab. 1968)
- Research and teaching
  - Bachelor, master, and PhD programme
  - ~70 tenures + lot of fellows

Parallel architecture lab. (current)
- 1 Full Prof. (M. Vanneschi)
- 1 Associate Prof. (M. Danelutto)
- 2 Researchers (M. Aldinucci, M. Coppola)
- 1 PostDoc (S. Campa)
- 2 PhD students (M. Meneghin, C. Bertolli),
- 2 senior engineers (M. Torquati, R. Ravazzolo)
- 4 junior engineers + several master students (in thesis)
Participation in Projects  
(1997-2007)

Ongoing

- IN.SY.EME (MIUR-IT FIRB) Integrated System for Emergency  - Jul. 2007, 36 m
- FRIMP (Cassa di Risparmio di Pisa) Software for Network Processors Feb. 2007, 24 m
- VirtuaLinux (Eurotech SpA) Roboust Virtual Clutering - Nov. 2006, 6 m
- BEinGRID (EU-IP, 6th FP) The Grid infrastructure for the Retail Management Experiment - Jun. 2006, 18 m
- GridComp (EU-STREP, 6th FP) Grid Component Model - June 2006, 30 m
- CoreGrid (EU-Network of Excellence, 6th FP): Foundations, Software Infrastructures and Applications for large scale distributed, Grid and Peer-to-Peer Technologies - 2004, 48 m

Completed

- Grid.it (MIUR FIRB) 2003 - 2006
- GridCoord (EU-Special Action, 6th FP) 2004 - 2006
- Vigoni Pisa-Berlino/Muenster (Exchange Programme) 2003 - 2005
- SAIB (Ricerca Industriale MIUR) 2002 - 2004
- Law 449/97 year 2000 (strategic projects MIUR-CNR) 2002 - 2004
- Law 449/97 year 1999 (strategic projects MIUR-CNR) 2002 - 2004
- ASI-PQE2000 (MIUR) 2001- 2002
- Agenzia2000 (MIUR) 2000-2002
- Vigoni Pisa-Passau (Exchange Programme) 1998 - 2000
- MOSAICO (MIUR 40%) 1998 - 2000
Scientific Productivity of the Lab (1997-2007)

- **Research & dissemination**
  - 21 intl. journals (8 A-class), 35 intl. conferences (20 A-class), 26 intl. workshops & symposium, 12 parts of books, served as editors for several journal and books, 2 large conferences organised (400+ attendees), several invited talks

- **Software (open source & copyrighted)**
  - 2 full programming environments for parallel languages
    - with language compiler: SkiE, ASSIST
  - several libraries for parallel programming
    - on top of Java, C, C++, Fortran, MPI, ACE, sockets, shmem, ...
  - servers and applications
    - distributed shared memory & storage, web server farm, // datamining, ...
  - cluster virtualization, cluster robustness, storage virtualization
    - VirtuaLinux
CGM model key points

- Hierarchic model
  - Expressiveness
  - Structured composition

- Interactions among components
  - Collective/group
  - Configurable/programmable
  - Not only RPC, but also stream/event

- NF aspects and QoS control
  - Autonomic computing paradigm
Why Autonomic Computing

// programming & the grid

- concurrency exploitation, concurrent activities set up, mapping/scheduling, communication/synchronisation handling and data allocation, ...
- manage resources heterogeneity and unreliability, networks latency and bandwidth unsteadiness, resources topology and availability changes, firewalls, private networks, reservation and jobs schedulers, ...

... and a non trivial QoS for applications

not easy leveraging only on middleware

our approach:

generally high-level methodologies + tools
Autonomic Computing paradigm

**Monitor**
- Collect execution stats: machine load, service time, input/output queues lengths, ...

**Analyse**
- Instantiate performance models with monitored data, detect broken contract, in and in the case try to detect the cause of the problem

**Plan**
- Select a (predefined or user defined) strategy to re-convey the contract to validity. The strategy is actually a “program” using execute API

**Execute**
- Leverage on mechanism to apply the plan

**QoS data**

**Managed components**

**Manager**
- **Monitor**
- **Analyse**
- **Plan**
- **Execute**

- **broken contract**
- **next configuration**
Why skeletons 1/2

- Management is difficult
  - Application change along time (ADL not enough)
  - How “describe” functional, non-functional features and their inter-relations?
  - The low-level programming of component and its management is simply too complex

- Component reuse is already a problem
  - Specialising component yet more with management strategy would just worsen the problem
  - Especially if the component should be reverse engineered to be used (its behaviour may change along the run)
Why skeletons 2/2

- Skeletons represent patterns of parallel computations (expressed in GCM as graphs of components)
- Exploit the inherent skeleton semantics
  - thus, restrict the general case of skeleton assembly
  - graph of any component → parametric networks of components exhibiting a given property
  - enough general to enable reuse
  - enough restricted to predetermine management strategies
- Can be enforced with additional requirements
  - E.g.: Any adaptation does not change the functional semantics
**Behavioural Skeletons idea**

- Represent an evolution of the algorithmic skeleton concept for component management
  - abstract parametric paradigms of component assembly
  - specialized to solve one or more management goals
    - self-configuration/optimization/healing/protection.
- 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
Behavioural Skeletons Properties

- Expose a description of its functional behaviour
- Establish a parametric orchestration schema of inner components
- May carry constraints that inner components are required to comply with
- May carry a number of pre-defined plans aiming to cope with a given self-management goal
- Carry an implementation (they are factories)
Be-Skeletons families

- Functional Replication
  - Farm/parameter sweep (self-optimization)
  - Simple 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
Functional replication

- **Farm**
  - S = unicast, C = from_any, W = stateless inner component

- **Data Parallel**
  - S = scatter, C = gather, W = stateless inner component

- **Fault-tolerant Active Replication**
  - S = broadcast, C = get_one_in_a_set, W = stateless inner ...
**Functional replication**

**Functional behaviour description**
(orchestration)

\[
\text{system}(\text{data}, S, G, W, \text{in}, \text{out}, N) \triangleq \\
S(\text{data}, \text{in}) | (| i : 1 \leq i \leq N : W_i(\text{in}_i, \text{out}_i)) | C(\text{out})
\]

\[
W_i(\text{in}_i, \text{out}_i) \triangleq \\
\text{in}_i.\text{get} > tk > \text{process}(tk) > r > (\text{out}_i.\text{put}(r) | W_i(\text{in}_i, \text{out}_i))
\]

- Meant to parametrically expose all allowed adaptation
- Any AM policy that does not change this semantics is **correct**
- As an example changing \( i \) in this schema is correct
- Functional semantics is invariant from \( i \), non-functional one is not
  (and changing \( i \) means changing the number of Ws for self-* purposes)
1. Choose a schema (.e.g. functional replication) ABC API is chosen accordingly

2. Choose an inner component (compliant to Be-Ske constraints)

3. Choose behavior of ports (e.g. unicast/from_any, scatter/gather)

4. Wire it in your application. Run it, then trigger adaptations

5. Possibly, automatize the process with a Manager

ABC = Autonomic Behaviour Controller (implements mechanisms)
AM = Autonomic Manager (implements policies)
B/LC = Binding + Lifecycle Controller
Farm example (Mandelbroot)

- change // degree
- new contract (e.g. Ts<k)
- get_service_time
- raise "contract violation"
- unicast
- from_any

ABC

screen
output
mandel
broot
mandel
broot
mandel
broot
ABC
lines
gen
S
C
mandel
broot
mandel
broot
mandel
broot
farm

unicast from_any
get_service_time
change // degree
raise "contract violation"
new contract (e.g. Ts<k)
Grid programming with components: an advanced COMPonent platform for an effective invisible grid
Not just farm (i.e. param sweep)

- Many other skeletons already developed for GCM
  - some mentioned before
- Easy extendible to stateful variants
  - imposing inner component expose NF ports for state access
- Policies not discussed here
  - expressed with a when-event-if-cond-then-action list of rules
  - some exist, work ongoing...
Typical Log of a Run (Explained)

- **Throughput (tasks/s)**
- **Avg. farm throughput**
- **QoS contract**
- **N. of PEs**
- **Time (minutes)**
- **N. of workers**
- **N. of PEs with artificial load**

The diagram shows the changes in throughput, number of PEs, and time over the course of a run. The QoS contract is indicated by a dotted line, and the past and future directions are marked on the graph.
new workers are mapped on empty nodes

new workers are mapped on nodes already running other instances of the same component

Overheads

![Graph showing Overheads over N. of workers](image)
**Proactive/Java Appears quite heavywight w.r.t. other approaches**

ASSIST/C++ overheads (ms)


<table>
<thead>
<tr>
<th>parmod kind</th>
<th>Data-parallel (with shared state)</th>
<th>Farm (without shared state)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add PEs</td>
<td>remove PEs</td>
</tr>
<tr>
<td>reconf. kind</td>
<td></td>
<td></td>
</tr>
<tr>
<td># of PEs involved</td>
<td>1→2 2→4 4→8</td>
<td>2→1 4→2 8→4</td>
</tr>
<tr>
<td>$R_l$ on-barrier</td>
<td>1.2 1.6 2.3</td>
<td>0.8 1.4 3.7</td>
</tr>
<tr>
<td>$R_l$ on-stream-item</td>
<td>4.7 12.0 33.9</td>
<td>3.9 6.5 19.1</td>
</tr>
<tr>
<td>$R_t$</td>
<td>24.4 30.5 36.6</td>
<td>21.2 35.3 43.5</td>
</tr>
</tbody>
</table>
Proactive Communication Time (Int)

Communication time

Communication Bandwidth (Theoretical 12800 KB/s)
Variations and Flavours

or in general ...

RPC or streaming data dependencies

and even more ...

RPC or streaming data dependencies
Abstracting Out Variants

- \( n \) client and \( y \) server ports
- synchronous and/or asynchronous
- stream and/or RPC
- programmable, possibly nondeterministic, relations among ports
  - wait for an item on port_\( A \) and/or one item on port_\( B \)
  - in general, any CSP expression

But ... be careful, this is the ASSIST model

- all features described above + distributed membrane + autonomicity, QoS contracts, limited hierarchy depth (i.e. 2)
- sophisticated C++ implementation, language not easy to modify

GCM should be enough expressive and not too complex

- we consider ASSIST as the complexity asymptote
Conclusions

 Behavioural Skeletons

 - templates with built-in management for the App designer
 - methodology for the skeleton designer
   - management can be changed/refined
   - just prove your own management is correct against skeleton functional description
 - can be freely mixed with standard GCM components
   - because once instanced, they are standard

 Already implemented on GCM

 - not happy about GCM runtime performances (can be improved)
   - We also implemented in ASSIST with different performances