Behavioural Specification and Verification for Compositional Grid Components

T. Barros, L. Henrio, E. Madelaine

www.inria.fr/oasis/
OASIS Team, INRIA -- CNRS - I3S -- Univ. of Nice Sophia-Antipolis
DCC, University of Chili
University Of Westminster

GRIDS@Work, CoreGRID WP3 Workshop, Oct. 14 2005
Agenda

• Context & Motivation
  • Components and safe composition
  • Fractive
• Building Behaviour models
• Checking Properties
• Conclusion & Perspectives
**Software Components**

**Definition:**
Software modules, composable, with well-defined interfaces, well-defined black box behaviour

**Features:**

1. **Encapsulation**
   - Black boxes, offered and required services

2. **Composition**
   - Design of complex systems, hierarchical organization into sub-systems

3. **Separate administration**
   - Architecture Description Language (ADL), administration components

4. **Distribution**
   - Interaction at interfaces through asynchronous method calls
**Behaviour specification and Safe composition**

**Definition**:  
Build reliable application using / re-using standard components  
Build reliable components from the composition of smaller pieces  
Component paradigm : only observe activity at interfaces.  
Deadlock freeness, progress/termination, safety and liveness.

**Scenarios**:
- **Build** complex systems from off-the-shelf components  
- **Check** protocol compatibility between sub-components  
- Check compatibility of sub-components behaviour with **deployment**  
- Check correctness of the **replacement** (update) of a sub-part of a running application  
- Take into account distribution and **asynchrony**.
Fractive’s components

Fractive = FRACTAL Implementation using ProActive

Fractal features:
• Hierarchical Components Model
• ADL description (Fractal’s XML Schema/DTD)
• Separation of functionality / management

Proactive features:
• Distributed components (from distributed objects)
• Asynchronous communication (non-blocking)
• Strong Formal Semantics => properties and guarantees
Fractal’s Components
**Fractive : Active objects**

- A ag = newActive ("A", [...], VirtualNode)
- V v1 = ag.foo (param);
- V v2 = ag.bar (param);
- ... v1.bar(); //Wait-By-Necessity
Wait-By-Necessity: First Class Futures

Futures are Global Single-Assignment Variables

\[ V = b.bar() \]

\[ c.\text{gee}(V) \]
First-Class Futures and Components

Asynchronous method calls with full-fledge Wait-By-Necessity

Non-blocking method calls

value of A

getAandB() — getA() — getB()

value of B
Fractal example

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE .... >
<definition name="components.System">
  <component ... </component>

  <component name="Alarm">
    <interface name="alarm" role="server"
      signature="components.AlarmInterface"/>
    <content class="components.Alarm"> </content>
    <behaviour file="AlarmBehav"
      format="FC2Param"/>
  </component>

  <binding client="BufferSystem.alarm"
    server="Alarm.alarm"/>
</definition>
Agenda

- Context & Motivation
- Building Behaviour Models
  - Formal Model
  - Configuration and Introspection
  - Asynchrony
- Checking Properties
- Conclusion & Perspectives
Behaviour: Parameterized Networks of synchronised Transitions Systems

Abstractions and Correctness

Program semantics \((1)\) => Behaviour Model \((2)\) => Finite Model

(1) user-specified abstract interpretation

(2) In the Value Passing case:
For a given formula,
define an abstract representation from a finite partition of the value domains.
\(\Rightarrow\) Preservation of safety and liveness properties [Cleaveland & Riely 93]

(2) For Families of processes
No hope for a similar generic result (but many results for specific topologies).
E.g. many reachability properties on parameterized topologies of processes deeply
requires induction reasoning and interactive theorem proving techniques.
Practical approach:
explore small finite configurations in a “bug search” fashion.
use state-of-the art “infinite systems” techniques for decidable domains when available
Build Fractive Behavioural Models Principles

Compositionality
• Reasonning at each separate composition level

Functional behaviour is known
• Given by the user
• Obtained by static analysis (primitive components, e.g. ProActive active objects)
• Computed from lower level…

Non functional behaviour automatically added from the component’s ADL
• Automata within a synchronisation network
• Incorporate controllers for control/management Fractal features, and for asynchronous communication management

Build the product, Hide, Minimize…. 
Build Fractive Behavioural Models

1) add non-functional controls
   = one LTS for each internal or sub-component interface
2) Asynchronous Membrane for primitive components
   = request queues, future proxies

3) ... and also for composites components
Result: The Static Automaton
(technical details in FACS’2005)

Deployment Automaton:

Static Automaton = (Controller || Deployment) + hiding & minimisation

Fine Tuning = Specify different hiding sets, depending on the properties we want to prove:
- deployment phase
- functional phase
- topology-preserving transformations
Agenda

• Context & Motivation
• Behaviour models
• Checking Properties
  • Tune model generation to various property classes
  • Functional and management interactions
  • Reconfiguration
• Conclusion & Perspectives
Behaviour correctness
(from the user point of view)

Initial Composition
- Generic properties:
  - successful deployment, presence of errors, deadlock freeness
- User Requirements expressed as temporal formulas

Reconfiguration (preserving or changing the components structure)
- Preservation of properties (aka service interaction)
- New properties (new features)

Compositionality
- The Static Automaton, after hiding/minimization, is the functional behaviour used at next level of composition
  - to be compared with the specification
Properties Verification (ACTL)

Deployment (on the Static Automaton with successful synchronisation visible)

- The deployment is always successful

\[ A(true \lor \neg true) \]

- Reachability of Errors during deployment

\[ AG_{true}[OE]false \]
Functional behaviour (on the Static Automaton)

- Get from the buffer eventually gives an answer

\[ [\text{true}^* \cdot \text{Request}()] \mu X. (\langle \text{true} \rangle > \text{true} \land \lbrack \neg \text{Response}() \rbrack X) \]

(inevitability of the Response)
Properties Verification
(regular $\mu$-calculus)

Functional properties under reconfiguration (respecting the topology)

- Future update (asynchronous result messages) independent of life-cycle or binding reconfigurations

- E.g:

  \[ [ \text{true}^\ast.\text{Request()} ] \mu X. (\langle \text{true} \rangle \text{true} \land [\neg \text{Response()} ] X) \]

  Proved on an Extended Static Automaton allowing the following control operations:

  $\text{?unbind}(C.E_b, B.E_g) \quad \text{?stop}(C)$
Structural Transformations (ongoing work)

Scenario:
Running application, Need to Replace/Update one sub-component
Check the protocol compatibility before replacement

Principle:
Use the formal Behaviour Specification
Identify states of the application behaviour model in which the transformation is possible,
… compute the corresponding states after transformation.
Use the merged automaton to check properties.

Benefits:
Identifies prerequisite and rules for safe transformation.
Agenda

- Context & Motivation
- Behaviour models
- Checking Properties
- Related Work, Conclusion & Perspectives
Related Work

Semantics :
- π-calculus, M-calculus, Kell calculus

From Process Algebras to Components
- Inherit from PA semantics : LTSs, (pre-) congruences
- Processes, Connectors, and CSP refinement : Wright
- Hierarchical composition, weak bisimulation, Buchi automata : Darwin

Sofa
- Frame (spec) vs. Architecture (implementation) compliance relation based on traces
- Hierarchical construction through parallel composition
- Detection of errors: bad activity, no activity and divergence

Behavioural Contracts (e.g. Carrez et al.)
- Behavioural typing (CSP like specification)
- Interface behavioural-type compatibility (decidable) and components contract compliance (non decidable).

No other work to my knowledge dealing with functionality + management (even in the synchronous case)
Ongoing work

Tool set:
- Code analysis (prototype, partial)
- Model generation (prototype, soon available)
- Interactions with model-checking and verification tools (available)

Expression of Properties:
- Pattern language specific to Grid Application

Extensions:
- Group communication

Perspectives:
- New verification tools (infinite state classes)
- "Safe by construction" programming style

Supported by FIACRE

An ACI-Security action of the French research ministry
Conclusions

• Model for the behaviour of hierarchical components
• Automatic Construction of the control parts for components management and for asynchronous communications
• Verification of properties in different phases

Asynchrony is essential for large scale grid applications (global efficiency, fewer deadlocks), but brings in new difficulties. Tools are strongly needed to help the developer.

• Implementation of a prototype tool for model construction, using standard model-checking tools for composition and verification of properties

Papers, Use-cases and Tools at:  http://www-sop.inria.fr/Vercors
Fractive’s implementation choices

Hierarchical start/stop
While stopped only non-functional request are served.
No path between functional and non-functional interfaces
ProActive and (De Facto) Standards

- RMI, RMI-Ibis, Jini, HTTP
- rsh, ssh, scp
- Globus GTx, sshGSI, Unicore, EGEE gLite
- LSF, PBS, OAR, Sun Grid Engine
- Fractal Components
- Web Services
- OSGi
Fractive Behavioural Models
Fractive implementation

Primitive component => Active object
Composite membrane => Active object
Previous work: ProActive behavioural models
(presented at Forte 2004)

Fractive composites
Properties Verification
(regular $\mu$-calculus)

Functional Properties under reconfiguration
• reconfiguration actions are allowed after deployment
Structural Transformations

- Controller of the new structure
- Action mapping
- Identify states in the original controller where transformation is possible (set $T$)
- "Connect" both controllers from $T$ through the transformation

$\tau(T)'$ is the image of $\tau$

The new behaviour is the reachable parts from $T'$ (using the action mapping)
Structural transformations

- No errors during functional phase

Synchronous former example

\[ AG_{true}[O_E] false \]

- `foo(A.l_c,B2.l_p) -> log(C,l_{log},Logger,l_{log}) -> ERROR_UNBOUND_ERI(B2,l_{log})`