An Evolving Hierarchical & Modular Approach to Resilient Software

Fernando J. Barros
University of Coimbra
barros@dei.uc.pt

Introduction

- Hierarchical and modular software is a sound paradigm to achieve resilient software
- Modularity provides the basic construct to identify faulty software units
  - Software units have well defined input and output interfaces
  - A faulty modular software unit can be replaced by a new version
    - enables fault correction

History

- The concepts of Hierarchical and Modular Systems design was formally introduced in the 60' in General Systems Theory (Wymore)
- Simula introduced OOP but not modularity
- Modular Simulation formalisms were created in the 70's Static structure formalisms (Zeigler 76)
- A formal definition (Semantics) of a Discrete Event Dynamic Structure Formalism was introduced in 90's (Barros 95)

History

- General Systems Theory Based Formalisms
  - Hierarchical
  - Modular
  - Timed-based
  - Deterministic
    - Deterministic simulations of stochastic systems
    - Simultaneous events
  - Dynamic Topologies
    - Mobility as a particular case
  - Closure under the coupling operation
    - Uniform handling of both basic and complex entities
  - Asynchronous
  - But, … offers an **awkward** model of programming (not suitable for software engineering)

History

- Connectons developed in 90s
  - A formalism based on Systems Theory
    - Keeps key features, but no timed-systems
    - Dynamic topology
    - *Ad-hoc* changes
    - Hierarchical Mobility
  - Based on the request/reply paradigm (OOP)
    - A model used in many languages: ST, C++, Java, …
    - Web services
  - Desmos implementation (Smalltalk)

Basic Connecton

$$M = \left(inGates, \{inSign_g\}, S, s_0, \{a_g\}, outGates, \{outSign_k\}, \{outFunction_k\}\right)$$

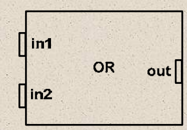
$inGates$, set of input gates
$inSign_g$, input-output signature $\forall g \in inGates$
S, set of states
s_0, initial state
a_g, an action for $\forall\ g \in inGates$
$outGates$, set of output gates
$outSign_k$, output-to-input signature $\forall\ k \in outGates$
$outFunction_k$, output function $\forall\ k \in outGates$

Basic Connecton

- An input signature is a tuple containing the range set of the incoming parameters and the range set of outgoing parameters
  - If input gate *g* receives real values **R**, and responds by sending integer values **I**, its input signature is given by $inSign_g$ = (**R**, **I**)
- An output signature is a tuple containing the range set of the outgoing parameters and the range set of incoming parameters
- The function a_g on input gate *g* of signature (I_g, O_g) is expressed by
  - $a_g: S \times I_g \rightarrow S \times O_g$
  - Actions correspond to methods in the object paradigm
- Output functions convert the set of values received by an output gate
  - Useful when several channels are linked to an output gate

7

OR Connecton



OR = ({out}, {(ϕ, B)}, {}, ϕ, {a_{out}}, {(in1, in2)}, {(ϕ, B)}, {(ϕ, B)})

8

Ensemble Connecton

$$E = \big(inGates, \{inSign_g\}, \{inFunction_g\}, \varepsilon, M_\varepsilon, outGates, \{outSign_k\}, \{outFunction_k\}\big)$$

inGates, set of ensemble input gates
$inSign_g$, input-output signature $\forall\ g \in inGates$
$inFunction_g$, input-function $\forall\ g \in inGates$
ε, ensemble executive
M_ε, model of the executive
outGates, set of ensemble output gates
$outSign_k$, output-to-input signature $\forall\ k \in outGates$
$outFunction_k$, output function $\forall\ k \in outGates$

9

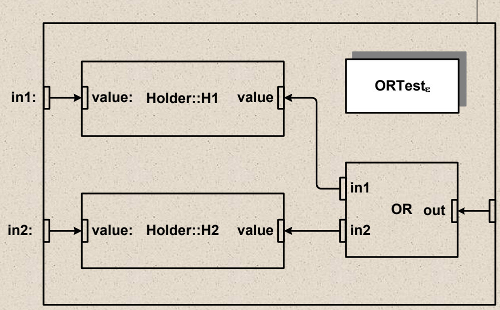
Executive Model

$$M_\varepsilon = \big(inGates, \{inSign_g\}, S, s_0, \{a_g\}, \sigma, \Sigma^*, outGates, \{outSign_k\}, \{outFunction_k\}\big)$$

$\sigma: S \rightarrow \Sigma^*$, structure function
$\Sigma = \big(C, \{M_c\}, L, \Xi\big), \forall\ \Sigma \in \Sigma^*$
C, set of connectons
M_c, model of each connecton $\forall\ c \in C$
L, set of channels
Ξ, order function

10

OR Test



11

OR Test

$$M_{ORTest} = \big(inGates, \{inSign_g\}, \varepsilon, M_\varepsilon, \{\}, \{\}, \{\}\big)$$

inGates = {{in1:}, {in2:}, {out}}
inSign = {(B, ϕ), (B, ϕ), (ϕ, B)}
$M_\varepsilon = (\{s_{0,\varepsilon}\}, s_{0,\varepsilon}, \sigma, \Sigma\text{^})$
$\sigma(s_{0,\varepsilon}) = (C, \{M_c\}, L)$
C = {H1, H2, OR}
$M_c = \{M_{H1}, M_{H2}, M_{OR}\}$
L = {((ORTest, {in1:}), (H1, {value:})),
((ORTest, {in2:}), (H2,{value:})),
((OR, {in1}), (H1, {value})),
((OR, {in2}), (H2, {value})),
((ORTest, {out}), (OR, {out}))}

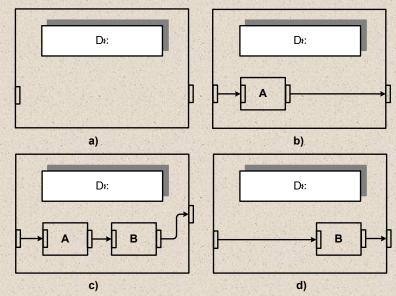
12

Structural Changes

- Topology adaptation has been subject to research in different areas like software engineering and general systems theory (simulation)
- The ability to change dynamically a running entity has been regarded as a powerful construct to build self-adaptive systems
- Methodologies supporting topology adaptation enable a representation with structural similarity
  - easier to develop and to maintain components

13

Structural Changes



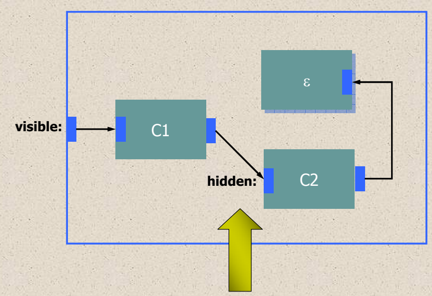
14

Hierarchical Mobility

- The use of hierarchical components in system representation brings a new problem not present in non-hierarchical systems
- Hierarchical components hide their inner constitution from the outside
  - How to expose inner components without violating encapsulation?
  - A system accessing the global software topology could modify the inner structure of any software unit but it will violate encapsulate
- Solutions proposed in systems theory and distributed systems, involve the use of mobile components
  - Components that can be transferred between two hierarchical components
- Inside an ensemble, a mobile component has access to the inner interface of a hierarchical component
  - No violation of encapsulation
- After visiting an ensemble, a mobile component can return with the gathered information

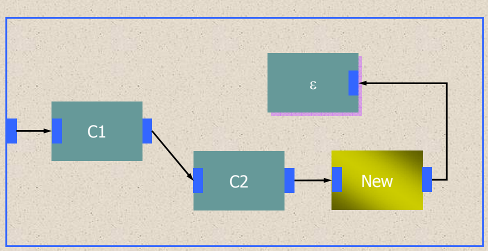
15

Hierarchical Mobility



16

Hierarchical Mobility



17

Hierarchical Mobility

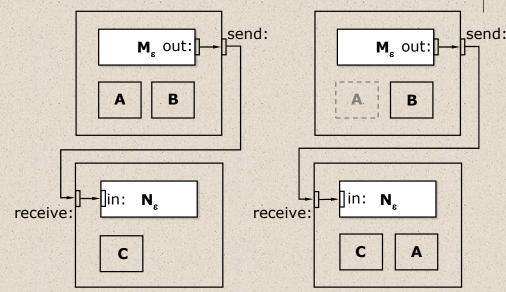
- A mobile component can be used to *extend* the interface of a hierarchical component
- A mobile component can be used to modify/fix the behavior of a hierarchical component
- A mobile component can be employed in a local reconfiguration bringing new behavior to an ensemble
  - The mobile component may become permanently part of the visited ensemble

18

Hierarchical Mobility

- Mobility requires the capability to remove a component from an ensemble and the ability to transmit it to another hierarchical component
- Transmission is achieved by message passing
- The visited ensemble needs to add the mobile component and to establish new links between the existing Connectons and the visiting one

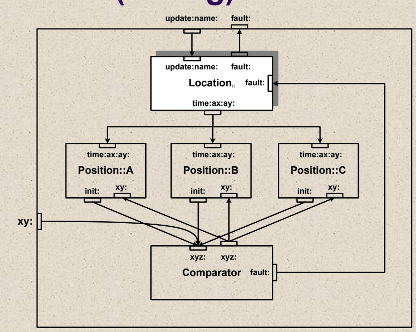
Hierarchical Mobility



Desmos Primitives

- **add:** aName **model:** aModel
  - adds to the ensemble a connecton named *aName* and associate it with model *aModel*
- **addConnecton:** aConnecton
  - adds an existing connecton
- **remove:** aConnecton
  - removes a connecton
- **link: aName gate: aGate to: bName gate: bGate**
  - links a Connecton named **aName** gate **aGate** to gate **bGate** of Connecton named **bName**
- **unlink: aName gate: aGate from: bName gate: bGate**
  - unlinks a Connecton named **aName** gate **aGate** from gate **bGate** of Connecton **bName**

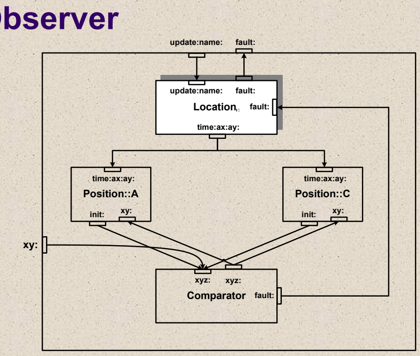
Observer(Voting)



Observer

```
Location>>structure
    super structure.
    "Adds redundant Position connectons"
    self add: #A model: Position.
    self add: #B model: Position.
    self add: #C model: Position.
    "Adds fault detector"
    self add: #CP model: Comparator.
    "Link definition"
    self link: #Network gate: #update:name: to: #Executive gate: #update:name:.
    self link: #Executive gate: #fault: to: #Network gate: #fault:.
    self link: #Executive gate: #time:ax:ay: to: #A gate: #time:ax:ay:.
    self link: #Executive gate: #time:ax:ay: to: #B gate: #time:ax:ay:.
    self link: #Executive gate: #time:ax:ay: to: #C gate: #time:ax:ay:.
    self link: #CP gate: #fault: to: #Executive gate: #fault:.
    "Reverse filters map 3D to 2D coordinates"
    self link: #Network gate: #xy: to: #CP gate: #xyz: rFilter: [:xyz| xyz toXY].
    self link: #A gate: #init: to: #CP gate: #xyz: rFilter: [:xyz| xyz toXY].
    self link: #B gate: #init: to: #CP gate: #xyz: rFilter: [:xyz| xyz toXY].
    self link: #B gate: #init: to: #CP gate: #xyz: rFilter: [:xyz| xyz toXY].
    "Reverse filters map 2D to 3D coordinates"
    self link: #CP gate: #xyz: to: #A gate: #xy: rFilter: [:xy| xy @ 0 @ #A].
    self link: #CP gate: #xyz: to: #B gate: #xy: rFilter: [:xy| xy @ 0 @ #B].
    self link: #CP gate: #xyz: to: #C gate: #xy: rFilter: [:xy| xy @ 0 @ #C].
```


Observer

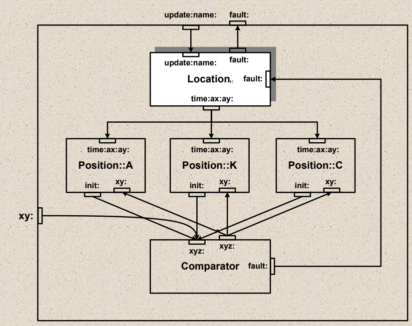


Observer

```
LocationExecutive>>fault: aName
    |faulty|
    faulty := self remove: aName.  "Removes connecton aName and all its links"
    out fault: faulty.             "Sends faulty as a mobile connecton“

LocationExecutive>>update: aPosition name: aName
    self add: aPosition name: aName.   "Adds a mobile connecton"
    self link: #Executive gate: #time:ax:ay: to: aName gate: #time:ax:ay:.
    self link: aName gate: #init: to: #CP gate: #x:y:z: rFilter: [:xyz| xyz asXY].
    self link: #CP gate: #xyz: to: aName gate: #xy: rFilter: [:xyz| xy @ 0 @ aName].
```


Observer



Conclusions

- We propose an approach to the representation of resilient software based on modular software units.
- Modularity enables the identification of faulty software units and their replacement with improved versions.
- Hierarchical mobility provides a sound construct to bring the updated version of faulty units, while keeping the encapsulation of hierarchical software.
- Hierarchical mobility enables online error correction while keeping the software running.