MUSA

A gentle introduction
MUSA
(Middleware for User-driven Service Adaptation)

• It is a general-purpose middleware for executing business processes
• On-demand Service Composition
• Dynamic orchestration of services
• Run-time goal monitoring
• Self-adaptation

• Current Version:
  https://github.com/icar-aose/musa_2
A Vision: Separating WHAT and HOW

WHAT (goal spec)

- WHEN pending(Meeting)
  AND meeting datetime(DT)
  AND attendee(Meeting,A)
  THE system SHALL
  PRODUCE
  notified(A,Meeting,DT)

- MUSA RUNNING SYSTEM

- bridge

HOW (capabilities)

- PROPOSAL MAIL SENDER
- COLLECT_MAIL_RESPONSES
- GOOGLECALENDARCHECK
The State of the World

• Real environments are Dynamic

• A state of the world ($W^t$) captures the current “state of affair”,
  – i.e. it is a symbolic description of the elements of interests in the environment.
  – Formally, it is a semantically coherent set of propositions

• Closed-world assumption: everything is not explicitly declared is assumed to be false.
Capabilities and Goals

• A Capability abstracts “what the system is able to do”

• A Goal abstracts a requirement, i.e. “what the system is expected to behave”
Map of the Elements

- **SYMBOLIC LEVEL**
  - SYSTEM CAPABILITIES
    - SERVICES
      - wraps
  - System Configuration
    - models
  - System Behavior
    - simulate

- **PROPERTY SPECIFICATION**
  - represented by

- **SYSTEM BEHAVIOR**
  - PMR
    - addresses
      - incrementally builds
  - USER'S GOALS
    - are translated into

- **GOALS MODEL**
  - $\mathcal{L}(C) \subseteq \mathcal{L}(G)$

- **MODEL CHECKER**
The MUSA 2.0 Architecture

• MUSA is programmed as a multi-agent system

• Technology:
  • Jason: http://jason.sourceforge.net/wp/
  • Jacamo: http://jacamo.sourceforge.net
    – Cartago: for programming environment artifacts
    – A choice we done is to move many algorithm inside a Java artifact
  • Tweety: http://tweetyproject.org/doc/index.html
    – A note: tweety contains a Conditional Logic and a Probabilistic Conditional Logic libraries
    – Example: "A usually implies B with probability p"
Part I: The *problem* exploration
Temporal Logic Goals

Goals

• Se il Generatore Principale si guasta bisogna accendere il Generatore Ausiliario

• Il Carico 1 deve essere sempre alimentato

• Il Carico 2 deve essere alimentato finché è acceso il Motore 2

• Se si verifica un incendio bisogna staccare l’alimentazione al Carico 2 e accendere il sistema antincendio

• $G (\text{off}(\text{genPrin}) \iff \text{on}(\text{genAus}))$

• $G (\text{on}(c1))$

• $G (\text{on}(c2) \cup \text{off}(m2))$

• $G( \text{verified}(\text{inc}) \implies (X \text{off}(c2) \land F \text{on}(\text{antiInc})) )$
LTL to PetriNet

• LTL permette di modellare il tempo tramite una successione infinita di Stati
• Una Rete di Petri è un linguaggio che permette la modellazione di un sistema distribuito
Planning Support and Goal Supervisor

 REQUIREMENTS  \[\rightarrow\]  LTL Goals

 \[\uparrow\]  Petri Net Model

Heuristic for Service Composition  \[\rightarrow\]  Supervisor Agent
We work under the assumption that the branching factor may be high, but finite. It is also worth noting that there may exist capabilities which evolution does not modify the portion of the state of the world that impact their preconditions. As a consequence, when preconditions are true, they remain true also after the capability execution: in these cases, the capability may be selected also at the next steps. An example is the capability \textit{visit city} which precondition is \textit{being at(palermo)} and the evolution function increments the value of \textit{visit time} in the predicate \textit{visited(palermo, visit time)} (not altering the predicate \textit{being at(palermo)}). For this reason we deal with potentially infinite-depth trees.

By using an exhaustive search algorithm (for instance the breadth-first search, see Figure 11), supposing a branching factor of $b$, the complexity order is $O(b + b^2 + b^3 + \ldots + b^d) = \Theta(b^d)$, where $d$ is the depth of the first discovered solution.

For limiting the complexity, we use a double strategy:

1. when $C + W_r = \emptyset$, it provides the preferences for selecting the node to expand, according to a depth-limited search,
2. otherwise, the algorithm is a depth-limited breadth-first search.

Algorithm 1 is the base search procedure that is explained in details in the following. The stack is a sorted list used to store all the possible capability sequence on work. Each item in the stack is in the form $h_{CS_r, evol, score}i$ where $r$ is the depth of the current node, $CS_r = (c_1, c_2, \ldots, c_r)$ is the capability sequence corresponding to the node and $evol = (W_1, W_1, W_2, \ldots, W_r)$ is the consequential state evolution. At the beginning stack is built by considering $C_00W_I$.

The solutions is an array (initially empty) where $max_solutions$ is the number of still empty slots. This array is used to collect the discovered sequences that satisfy the problem. Finally, the integer $max_depth$ is the max depth allowed in the search.

We work under the assumption that the branching factor may be high, but finite. It is also worth noting that there may exist capabilities which evolution does not modify the portion of the state of the world that impact their preconditions. As a consequence, when preconditions are true, they remain true also after the capability execution: in these cases, the capability may be selected also at the next steps. An example is the capability \textit{visit city} which precondition is \textit{being at} (\texttt{palermo}) and the evolution function \textit{increments the value of} \textit{visit time} in the predicate \textit{visited} (\texttt{palermo}, \textit{visit time}) (not altering the predicate \textit{being at} (\texttt{palermo})). For this reason we deal with potentially infinite-depth trees.

By using a exhaustive search algorithm (for instance the breadth-first search, see Figure 11), supposing a branching factor of $b$, the complexity order is $O(b + b^2 + b^3 + \ldots + b^d) = O(b^d)$, where $d$ is the depth of the first discovered solution.

For limiting the complexity, we use a double strategy:

1. When $C + W_{r6} = \_\_\_$, it provides the preferences for selecting the node to expand, according to a depth-limited search,
2. Otherwise, the algorithm is a depth-limited breadth-first search.

Algorithm 1 is the base search procedure that is explained in details in the following.

The stack is a sorted list used to store all the possible capability sequence on work. Each item in the stack is in the form $h_{CS_r, evol, score}$ where \texttt{r} is the depth of the current node, $CS_r = (c_1, c_2, \ldots, c_r)$ is the capability sequence corresponding to the node and $evol = (W_1, W_1, W_2, \ldots, W_r)$ is the consequential state evolution. At the beginning stack is built by considering $C_{00} W_I$.

The solutions is an array (initially empty) where $max solutions$ is the number of still empty slots. This array is used to collect the discovered sequences that satisfy the problem. Finally, the integer $max depth$ is the max depth allowed in the search.
We work under the assumption that the branching factor may be high, but finite. It is also worth noting that there may exist capabilities which evolution does not modify the portion of the state of the world that impact their preconditions. As a consequence, when preconditions are true, they remain true also after the capability execution: in these cases, the capability may be selected also at the next steps. An example is the capability visit city which precondition is being at(palermo) and the evolution function increments the value of visit time in the predicate visited(palermo, visit time) (not altering the predicate being at(palermo)). For this reason we deal with potentially infinite-depth trees.

By using an exhaustive search algorithm (for instance the breadth-first search, see Figure 11), supposing a branching factor of $b$, the complexity order is $O(b + b^2 + b^3 + \ldots + b^d) = O(b^d)$, where $d$ is the depth of the first discovered solution.

For limiting the complexity, we use a double strategy:

1. When $C + W = 6$, it provides the preferences for selecting the node to expand, according to a depth-limited search,
2. Otherwise, the algorithm is an depth-limited breadth-first search.

Algorithm 1 is the base search procedure that is explained in details in the following. The stack is a sorted list used to store all the possible capability sequence on work. Each item in the stack is in the form $h_{CS_r, evol, score}$ where $r$ is the depth of the current node, $CS_r = (c_1, c_2, \ldots, c_r)$ is the capability sequence corresponding to the node and evol $= (W_1, W_2, W_3, \ldots, W_r)$ is the consequential state evolution. At the beginning stack is built by considering $C_00W_I$.

The solutions is an array (initially empty) where max solutions is the number of still empty slots. This array is used to collect the discovered sequences that satisfy the problem. Finally, the integer max depth is the max depth allowed in the search.
We work under the assumption that the branching factor may be high, but finite. It is also worth noting that there may exist capabilities which evolution does not modify the portion of the state of the world that impact their preconditions. As a consequence, when preconditions are true, they remain true also after the capability execution: in these cases, the capability may be selected also at the next steps. An example is the capability
\[ \text{visit city} \]
which precondition is
\[ \text{being at( palermo)} \]
and the evolution function
\[ \text{increments the value of visit time in the predicate} \]
\[ \text{visited( palermo, visit time)} \]
(not altering the predicate \text{being at( palermo)}). For this reason we deal with potentially infinite-depth trees.

By using an exhaustive search algorithm (for instance the breadth-first search, see Figure 11), supposing a branching factor of \( b \), the complexity order is
\[ O(b + b^2 + b^3 + \ldots + b^d) = O(b^d) \]
where \( d \) is the depth of the first discovered solution.

For limiting the complexity, we use a double strategy:
1. When \( C + W = 6 \), it provides the preferences for selecting the node to expand, according to a depth-limited search,
2. Otherwise, the algorithm is a depth-limited breadth-first search.

To which one select for the subsequent iteration?
We work under the assumption that the branching factor may be high, but finite. It is also worth noting that there may exist capabilities which evolution does not modify the portion of the state of the world that impact their preconditions. As a consequence, when preconditions are true, they remain true also after the capability execution: in these cases, the capability may be selected also at the next steps. An example is the capability \textit{visit city} which precondition is \textit{being at(palermo)} and the evolution function increments the value of \textit{visit time} in the predicate \textit{visited(palermo, visit time)} (not altering the predicate \textit{being at(palermo)}). For this reason we deal with potentially infinite-depth trees.

By using a exhaustive search algorithm (for instance the breadth-first search, see Figure 11), supposing a branching factor of $b$, the complexity order is $O(b + b^2 + b^3 + \ldots + b^d) = O(b^d)$, where $d$ is the depth of the first discovered solution.

For limiting the complexity, we use a double strategy:

1. when $C + W = \frac{W_1}{6}$, it provides the preferences for selecting the node to expand, according to a depth-limited search,
2. otherwise, the algorithm is an depth-limited breadth-first search.

Algorithm 1 is the base search procedure that is explained in details in the following. The stack is a sorted list used to store all the possible capability sequence on work. Each item in the stack is in the form $h_{CS_r, evol, score}$ where $r$ is the depth of the current node, $CS_r = (c_1, c_2, \ldots, c_r)$ is the capability sequence corresponding to the node and $evol = (W_I, W_1, W_2, \ldots, W_r)$ is the consequential state evolution. At the beginning stack is built by considering $C_0W_I$.

The solutions is an array (initially empty) where $max solutions$ is the number of still empty slots. This array is used to collect the discovered sequences that satisfy the problem. Finally, the integer $max depth$ is the max depth allowed in the search.

To which one select for the subsequent iteration?
We work under the assumption that the branching factor may be high, but finite. It is also worth noting that there may exist capabilities which evolution does not modify the portion of the state of the world that impact their preconditions. As a consequence, when preconditions are true, they remain true also after the capability execution: in these cases, the capability may be selected also at the next steps. An example is the capability visit city which precondition is being at(palermo) and the evolution function increments the value of visit time in the predicate visited(palermo, visit time) (not altering the predicate being at(palermo)). For this reason we deal with potentially infinite-depth trees.

By using an exhaustive search algorithm (for instance the breadth-first search, see Figure 11), supposing a branching factor of $b$, the complexity order is $O(b + b^2 + b^3 + ... + b^d) = O(b^d)$, where $d$ is the depth of the first discovered solution.

For limiting the complexity, we use a double strategy:

1. when $C + W = 6$, it provides the preferences for selecting the node to expand, according to a depth-limited search,
2. otherwise, the algorithm is a depth-limited breadth-first search.

Algorithm 1 is the base search procedure that is explained in details in the following.

The stack is a sorted list used to store all the possible capability sequence on work. Each item in the stack is in the form $h_{CS_r, evol, score}$, where $r$ is the depth of the current node, $CS_r = (c_1, c_2, ..., c_r)$ is the capability sequence corresponding to the node and $evol = (W_1, W_1, W_2, ..., W_r)$ is the consequential state evolution. At the beginning stack is built by considering $C_00W_I$.

The solutions is an array (initially empty) where $max_solutions$ is the number of still empty slots. This array is used to collect the discovered sequences that satisfy the problem. Finally, the integer $max_depth$ is the max depth allowed in the search.

When the score is MAX then the state is marked as Solution.
Element: WTS

• The space exploration is stored as a World Transition System

• The WTS is a graph:
  – Nodes = State of the world
  – Edges = Capability used to change state of the world

• Worth noting that:
  – WTS has some special nodes (XOR)
  – WTS may contain loops (and probably will)
  – WTS has final nodes. The ones we like
The service composition is a distributed algorithm

• The graph manager agent is responsible of a shared Graph artifact

• Capabilities are distributed across many Explorer agents
  – An explorer may read the graph
  – An explorer generates possible “expansions” in autonomy
  – Expansions are stored as agent’s believes

• A periodic auction-protocol regulates the write-access permission to the graph
The Auction

a) selection of the highest scored node

b) generating new nodes

c). evaluating

and again a)
Element: A Solution

• Solutions are **trees**.
• Some cases to distinguish:
  – linear paths
  – paths which contains loops
  – presence of a xor node
Solutions are executable Workflows
The *spec manager agent* is responsible of “extracting” viable solutions from the WTS (during its construction).

The *case manager agent* has the responsibility to select the best solution among the available ones.
An Example: cloud app mashup
Part II: The *execution* of the solution (still work in progress)
HOW TO USE MUSA
(a user’s manual)
Three stakeholders

- **User**
  - Define Specifications
    - Functional
    - Non-Functional
  - Specification Management
    - Evolution
    - Injection/Retreat

- **Musa**
  - Domain Definition
    - Domain Assumptions Definition
    - General Configurations Set
  - Domain Configurations

- **Admin**
  - Abstract Capability Creation
    - <<include>>

- **Dev Team**
  - Browse Abstract Capability by Domain
    - <<include>>
  - Upload Concrete Capability for a Domain
    - Associate Abstract-Concrete
Musa Dashboard

Available at: http://aose.pa.icar.cnr.it:8080/musaGUI/index.jsp
1. Creating a Domain

- A *domain of interest* is an application field where MUSA can run

- A domain is the set of
  - Ontological Assumptions
  - Abstract Capabilities

<table>
<thead>
<tr>
<th>NAME</th>
<th>NOTES</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>dom name2</td>
<td>New Description</td>
<td>EDIT  DELETE  CONFIGURE  DEFINE ASSUMPTIONS  MANAGE ABSTRACT CAPABILITIES</td>
</tr>
<tr>
<td>new dom name</td>
<td>my descr</td>
<td>EDIT  DELETE  CONFIGURE  DEFINE ASSUMPTIONS  MANAGE ABSTRACT CAPABILITIES</td>
</tr>
<tr>
<td>test</td>
<td>domainin test</td>
<td>EDIT  DELETE  CONFIGURE  DEFINE ASSUMPTIONS  MANAGE ABSTRACT CAPABILITIES</td>
</tr>
<tr>
<td>ff</td>
<td>ff</td>
<td>EDIT  DELETE  CONFIGURE  DEFINE ASSUMPTIONS  MANAGE ABSTRACT CAPABILITIES</td>
</tr>
<tr>
<td>pppp0</td>
<td>pppp0</td>
<td>EDIT  DELETE  CONFIGURE  DEFINE ASSUMPTIONS  MANAGE ABSTRACT CAPABILITIES</td>
</tr>
</tbody>
</table>
Domain Assumption

• The objective is to provide invariants about the environment
• A domain assumption is a IF …. THEN rule
  – Premise => Conclusion
• Example
  – up(n2) :- up(n1) & closed(i17)
  – If a state of world contains the two statements \( up(n1) \) and \( closed(i17) \) then it is possible to deduct the new fact \( up(n2) \)
Abstract Capability

• The objective is to provide a high-level description of an Action the system may use for some purpose
• It is used to generalize categories of similar services
• An Abstract Capability provides:
  – Service Pre-conditions
  – A set of possible evolutions due to Service execution
  – [not yet implemented] Parameters/assumptions necessary for Service execution

• Example

```plaintext
CAPABILITY check_storehouse:
Parameters: an_order, a_user
Assumptions: order(an_order), user(a_user)
Pre-Condition: available(X) & order(X) & registered(X) & user(X)
EvolutionSet:
  AcceptableOrder: [add(accepted(an_order)), remove(available(an_order))],
  UnacceptableOrder: [add(refused(an_order)), remove(available(an_order))]
```
2. Providing Services

- A Service (concrete capability) is bound to an abstract capability
- The relation is 1-n (1 abstract, n concretes)

**CONCRETE CAPABILITIES (DOMAIN:domain2 )**

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATE</th>
<th>ABSTRACT</th>
<th>NOTES</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>gdf</td>
<td>unactive</td>
<td>test</td>
<td>gg</td>
<td>MODIFY</td>
</tr>
<tr>
<td>cap name4</td>
<td>unactive</td>
<td>test</td>
<td>sdfsf</td>
<td>MODIFY</td>
</tr>
<tr>
<td>cap name5</td>
<td>active</td>
<td>test</td>
<td>sdfsd</td>
<td>MODIFY</td>
</tr>
<tr>
<td>cap name5</td>
<td>unactive</td>
<td>test</td>
<td>sdgs</td>
<td>MODIFY</td>
</tr>
<tr>
<td>sfisd</td>
<td>unactive</td>
<td>test</td>
<td>sff</td>
<td>MODIFY</td>
</tr>
</tbody>
</table>
Concrete Capability

• The objective is to provide the “agentification” of a Service
  – A Worker Agent must be launched in MUSA workspace
  – This agent is able to properly invoke the service
• A Concrete Capability relates the worker agent to a specific domain via a specific Service
• Providing a concrete capability means specifying
  – Where the Agent runs (IP Address, Agent Name)
  – How to communicate with that agent (I/O Ports)
3. Specifying Requirements

- The user injects/retreats/modifies
  - Functional Requirements
  - Non-Functional Requirements
LTL Goal

• The objective is providing a description of how the system will behave (user’s expectations)
• The whole LTL syntax is admitted (Globally, Finally, Next, Until and Release)
• Example:
  – G(verified(a_fire) => X off(l2) && F on(the_fire_sys) )
    i.e.
  – Every time a fire occurs [on board] then the load l2 must be soon switched off and finally the load the_fire_sys must be switched on
4. MUSA identifies a set of solutions

• 4.1 The *Spec Manager Agent* translates Goals into Petri-Nets
• 4.2 The *Graph Manager Agent* creates a new empty WTS
• 4.3 *Explorer Agents* collaborate to build the WTS
• 4.4 The *Spec Manager Agent* extracts viable solutions
  – If at least a solution exists, then the Case Manager agent starts waiting for new Requests (see next slide)
Two different Behaviours

• **Case 1: 1 set of goals – 1 system execution**
  – This configuration is used when the system has a unique response to the goal injection
  – Examples: Shipboard Power System Reconfiguration, Smart Space System, Robotic Application

• **Case 2: 1 set of goals – n execution cases**
  – This configuration is preferable when the system may serve different users within the same application domain
  – Example: Smart Travel, Dynamic Workflow, Cloud Application Mashup, User-Intensive Exhibition Centre

• This is a specific property to be set for each domain
5. [work in Progress] New Case

• [only for the case 1-n]
  – The user requests a new case to be executed
    • This is generally done by a HTTP request
  – A *Proxy Agent* receives the HTTP message and forwards it to the Case Manager Agent that creates a new case

• The *Case Manager Agent* picks a solution and starts an orchestration society
  – The *Orchestration Society* is made of the *Worker Agents* who own the corresponding services (belonging to the solution)

• A *Goal Supervisor Agent* starts observing the environment via the supervisor petri-net model for detecting failures and goal violations

• The *Case Manager* handles Adaptation Signals by
  – Replacing the malfunctioning service
  – Selecting an alternative solution and changing the orchestration society consequently