

UNIVERSITY OF ROME 'LA SAPIENZA'  
ENGINEERING FACULTY



# “OPUS”

---

# Service Integration

Project Documentation

*Faculty of Engineering*

*Course in Computer Engineering*

*Authors: Balestra Concetta, Coccia Chiara, Colarullo Diego, Iachini Alberto, Qusa Hani*

*Promoter: Prof. Giuseppe De Giacomo*

**2009, 15th May**

2009, 15th May

## Index

Overview .....	3
Project Structure .....	4
Beginning Idea.....	4
Work Division .....	6
First Part	
Chapter 1: Abstraction Package.....	8
1.1 First Version .....	8
1.1.1 Transition Function.....	8
1.1.2 Read From File .....	8
1.1.3 Service .....	9
1.1.4 Cartesian Product.....	9
1.2 Second Version .....	11
1.2.1 State .....	11
1.2.2 Action .....	11
1.2.3 State Action .....	12
1.2.4 Transition Function.....	12
1.2.5 Convert Pdf.....	12
1.2.6 Read File.....	13
1.2.7 Service .....	14
1.2.8 Community .....	17
1.3 Third Version .....	21
Chapter 2: Final Release	
2.1 Package description.....	24
2.2 Implementation Choices.....	33
Second Part	
Overview .....	35
Chapter 1: Control Package Description.....	37
1.1 Simulation algorithm.....	37
1.2 Simulation Implementation .....	40
1.3 Orchestrator class .....	44
1.4 Package Version .....	48
Chapter 2: Creation of community states.....	49
2.1 Control Package v2.....	49
Third Part	
Chapter 1: Graphical user interface .....	52
1.1 Swing Components.....	52
2 Chapter 2: Panels .....	53
2.1 Available Panel and Target Panel.....	53
2.2 Cartesian ProductPanel.....	53
2.3 Orchestrator Panel and Execute Orchestrator Panel .....	53
2.4 The libraries JGraph and JGraphLayout.....	54
2.5 Configuration .....	54
Fourt Part	
Chapter1: Test Cases.....	56

## Overview

Our project is based on the idea to transport composition via simulation algorithm into a java program. The composition concept requires other notions like Transition Systems, Service, Community, Orchestrator and simulation. A transition system is a graphical representation of program behavior and, more in general, is a tuple  $T = \langle A, S, S^0, \delta, F \rangle$  where:

- ✓ A: set of action;
- ✓ S: set of state;
- ✓  $S^0$ : set of initial state ( $S^0 \subseteq S$ );
- ✓  $\delta$ : transition relation ( $\delta \subseteq S \times A \times S$ );
- ✓ F: set of final state ( $F \subseteq S$ ).

In a Transition System T some properties can be present like reachability, that represents the possibility to reach a node covering a path among T's nodes, and simulation between two Transition System T and S, for which T is copied by S. So, a state  $s_0$  of S is similar, or simply equivalent, to a state  $t_0$  of T if and only if there exists a simulation between the initial states  $s_0$  and  $t_0$ <sup>1</sup>.

A Service's behavior is a finite Transition System. Community represents a set of Services, that share implicitly a common set of actions and export their behavior using finite Transition System over this common set of actions. A client specifies needs as a Service behavior like a finite Transition System using the common set of actions of Community<sup>2</sup>. Target Service represents Client's actions and Available Services are the set of Community Services.

With our work we want to generate an Orchestrator on Community, that is able to do a mapping among Available Services. To build Orchestrator we need to implement an algorithm that uses simulation concept.

<sup>1</sup> Transition Systems and Bisimulation, Giuseppe De Giacomo - Service Integration A.A. 2008/09

<sup>2</sup> Composition via Simulation, Giuseppe De Giacomo - Service Integration A.A. 2008/09

---

2009, 15th May

## Project Structure

The idea to convert theoretical notions on a concrete java program has been developed first of all in a UML class diagram, where the most important classes have been described in a primordial and formal way. Then, our developing group decided to divide the work into three packages. Package's division is translated into a workgroup's division. This allowed us to work in parallel with different part of the project.

### Beginning Idea

Beginning UML description is the follow:

*Service:*

This class can be used to translate in a Java's format the graph of a service.

We memorize all the action, the present and next state in three String's array. The fields are:

- ✓ *actions*: array String;
- ✓ *presentStates*: array String;
- ✓ *nextStates*: array String.

The methods of this class are:

- ✓ *getActions()*: it returns the array of action;
- ✓ *getPresentStates()*: it returns the array of present state;
- ✓ *getNextStates()*: it return the array of next state;
- ✓ *getPresentState(String action)*: it takes an action and returns all the possible states from which this action is available.
- ✓ *getNextState(String action)*: it takes an action and returns all the possible states that are reachable performing this action.
- ✓ *setActions(String array actions)*: the array of the actions is set by the input.
- ✓ *setPresentStates(String array presentStates)*: the array of the presentStates is set by the input.
- ✓ *setNextStates( String array nextStates)*: the array of the nextStates is set by the input.
- ✓ *positionAction(String action)*: it returns the indexes of all the position in the actions' array of he input(NB this position are the same of the present and next state that can perform and are reachable by the input, in fact all the arrays(actions, presentStates and nextStates) of this class have the same dimension. This because a service can be non-deterministic and so, performing an action, we can achieve different next states).

---

2009, 15th May

### *ReadFileInput:*

This class contains only the method that read the file .dot and creates a service

The method is:

- ✓ *fromFile(String fileName)*: it returns the service present in file.

### *Orchestrator:*

This class creates the orchestrator from all the available services. The orchestrator is a service where all the states and action came from combination of states and actions of the available service. For this reason this class extends the Service one.

The methods are:

- ✓ *static createOrchestrator(array Service availableSevices)*: it creates the orchestrator from all the available services.

### *Simulation:*

This class verifies the simulation between the orchestrator and the target service.

The methods are:

- ✓ *verifySimulation(String action, String presentStateOrchestretor, String nextStateOrchestrator, array String actionsTarget, array String presentSatesTarget, array String nextStatesTarget)*: it returns a Boolean value that is 1 if the states of orchestrator are in simulation with one of the states in the target service.
- ✓ *storeSimulation(Orchestrator orchestratorGenerator, Service targetService)*: it returns a SpareMatrix where are stored the results of simulation.

This is a very primordial definition of our project's main classes and it doesn't care particularly on presentation part. For the visualization of algorithm's output we thought, initially, to realize within a java swing panel a view of transition system using jGraphT library integrated with a more understanding representation like textual one.

## Work Division

The three packages we worked with are:

- ✓ Presentation
- ✓ Control
- ✓ Abstraction

This choice comes from the implementation of PAC structure, for which the first level (Abstraction) contains all classes that represents the information managed by the application; the second level (Control) manage data flow between the other two package; the last one (Presentation) manages user interaction, so the input and output of application<sup>3</sup>. This division makes independent the three modules that implements different functionality of system, every package haven't any role for the others.

PAC structure brought us to a group division and made possible working in a separated way.

---

<sup>3</sup>

*PAC: Presentation – Abstraction - Control (ovvero l'Architettura di un'Applicazione Software), Claudio Di Ciccio, Massimo Mecella*

---

2009, 15th May

# **First Part:**

## **Abstraction Package**

## Chapter 1: Abstraction Package

In general, this part contains all classes that can represent a service, its states and actions and the relations among these structures. The most of the classes present in this package identify immutable object, therefore in these classes we don't do the override of equals and hashCode methods. In this way an object is equals to another if and only if it's the same object in the system. This choice was applied in last versions of abstraction package. The content of a service is taken from a text file written with an appropriate formalism. Every service, which we can identify like an available service, is used to fill the structures in AvailableService class. Furthermore, the Cartesian product is built on community's content. The result of this operation will be a new service.

### 1.1 First Version

The classes present in first version of abstraction package are:

- ✓ Transition Function
- ✓ Read From File
- ✓ Service
- ✓ Cartesian Product

#### *Transition Function*

Class with three type String fields representing presentState, action and nextState of a transition system. Methods are:

- ✓ **Constructor:** empty constructor and TransitionFunction(String presentState, String action, String nextState);
- ✓ **Override:** equals and toString;
- ✓ **Other methods:** get and set on presentState, action and nextState;

#### *Read From File*

Class used to build a service from a .dot file. Methods are:

- ✓ static boolean checkFile(String filename): it checks the .dot file to proof if this file was written in a right way;
- ✓ static Service fromFile(String filename): it returns the service containing information present in .dot file.

Our .dot formalism doesn't correspond to all constructs present in the dot language, in fact we foresee for our project a sub dot language. An example of file accepted by our application is:

```
digraph available{
    node [shape = doublecircle]; S0;
    node [shape = circle];
    S0 -> S1 [ label = "a" ];
    S1 -> S0 [ label = "c" ]; }
```



## Service

The fields of this class are: a list of transition function named states and a list of String that represent final states named finalStates. Methods are:

- ✓ **Constructor:** empty constructor;
- ✓ **Override:** toString;
- ✓ **Other methods:** we have all the methods to get, set, add and delete a transition function and a final states from the fields of this class, moreover we have methods to get and set present state, next state and action. Some peculiar methods are:

List<String> getState()

it returns the list of all service's states doing a union between present and next state, avoiding duplicate;

List<TransitionFunction> getActionFromState(String state)

it returns all the transition function with String state like for present state;

## Cartesian Product

Class used to build Cartesian product among available services. It returns a service. Methods are:

- ✓ Recursive method doing the Cartesian product among the available services

```
static Service executeProductStates(List<Service> availableService){
    Service states = new Service();
    if(availableService.size()==0) return states;
    if(availableService.size()>2){
        states = productService(first available Service, second available Service);
        remove first available service from the list;
        add the new service, that corresponds to Cartesian product between first two
        services of the list, at first position in available services list;
        remove second available service from the list;
        //do a recursive call
        states = executeProductStates(availableService);
    }
    else{
        if(availableService.size()==2)
            states = productService(first availableService, second availableService);

        else{
            //list size ==1
            states = first and only one availableService;
        }
    }
}
```

2009, 15th May

```
    } return states ; }
```

- ✓ Method doing the Cartesian product between two service

```
static Service productService(Service s1, Service s2){
    Service prod = new Service();
    List<String> finalState = product(s1.getFinalState(),s2.getFinalState());
    //give to new service the final state product
    prod.setFinalState(finalState);
    List<TransitionFunction> result = new ArrayList<TransitionFunction>();
    for(all the transitions i of first service)
        for(all the transitions j of second service){
            create four new transition functions resulting from this combination:
            

- s1 present state, s2 present state – s1 action – s1 next state, s2 present state;
- s1 present state, s2 next state – s1 action – s1 next state, s2 next state;
- s1 present state, s2 present state – s2 action – s1 present state, s2 next state;
- s1 next state, s2 present state – s2 action – s1 next state, s2 next state.


            Add this new transition function to list if they are not still present.
        }
    prod.setTransactionFunctionCollection(result);
    return prod;
}
```

- ✓ Method that execute Cartesian product between two String list

```
static List <String> product (List<String> l1, List<String> l2){
    List<String> prod
    for(all string i in l1)
        for(all string j in l2){
            prod.add(i.concat(j));
        }
    return prod;
}
```

## 1.2 Second Version

To improve the efficiency of methods that give to other packages the information on a service, we introduced some new fields in class Service and we built new classes to represent State and Action. These classes increase the modularity of program so that we manage State and Action separately from a Service. In the first version we used a structure of String, so in this release we changed String in Action or State opportunely. For instance, if in the future we will need to change the type of State we will modify, very quickly, only State class differently from previous version where we had to modify different files.

Furthermore, file in input are no more DOT file, but textual files written in a precise way. So, to create a new service, we had to give to service constructor a String, which content will be the information on new service.

The classes present in this package for second version are:

- ✓ State;
- ✓ Action;
- ✓ StateAction;
- ✓ TransitionFunction;
- ✓ ConvertPdf;
- ✓ ReadFile;
- ✓ Service;
- ✓ Community;
- ✓ CartesianProduct.

### *State*

This class represent a service's state. Its fields are a string that maintains the name of this state and a Boolean property isfinal. Methods are:

- ✓ **Constructor:** public State(String name) setting name in State class to the variable name and isfinal to false;
- ✓ **Override:** equals, hashCode and toString;
- ✓ **Other methods:** get and set on state name and isfinal property.

### *Action*

This class represents a service's action. Its field is a string that maintains the name of this action. Methods are:

- ✓ **Constructor:** public Action(String name) setting action name;
- ✓ **Override:** equals, hashCode and toString;
- ✓ **Other methods:** get and set on action name.

2009, 15th May

*State Action*

This is a support class used to maintaining information on pair state-action in service class. We are interest to know which states are linked by a specific pair of state-action. Its fields are a State presentState and an Action action. Methods are:

- ✓ **Constructor:** public StateAction(State presentState, Action action) setting presentState and action;
- ✓ **Override:** equals, hashCode and toString;
- ✓ **Other methods:** get and set on action and presentState;

*Transition Function*

In new version of this class we introduced action and state class instead of previous string fields. So the new fields are: State presentState, Action action and State nextState. Furthermore we insert into this class the override of hashCode method.

*Convert Pdf*

Our Application can read a PDF file. To do this we created ConvertPdf class. It is used by ReadFile class to convert the content of pdf in textual one. In this class is used a special java library named **PDFBox-0.7.3.jar**. Methods are

```
protected static String GetTextFromPdf(String filename)
throws IOException, NoClassDefFoundError, Exception{
    String contentFile="";
    PDDocument document=null;
    InputStream in=new FileInputStream(filename);
    StringWriter out=new StringWriter();
    //use of pdf parser with document in input
    PDFParser parser = new PDFParser(in);
    parser.parse();
    //give to new document the parser one
    document = parser.getPDDocument();
    //creation of a text stripper to take document content
    PDFTextStripper stripper = new PDFTextStripper();
    stripper.writeText(document,out);
    contentFile=out.toString();

    //close new document
    document.close();

    //return a string with file content
    return contentFile; }
```

The catch exception has left to Read File class.

---

2009, 15th May

## *Read File*

Class used to make the parse of document in input. These documents could be:

- .txt;
- .aut;
- .doc, .docx;
- .pdf.

To be accepted by our application a file has to be written in this way:

```
transition:
presentStateName1-action-nextStateName1;
...
presentStateNameN-actionM-nextStateNameN;
final:
state1Name;
...
stateNName.
```

The first line of a document should be `transition: .` After this line the user writes all the transition of a service with the accuracy to don't put any blank space in `presentStateName1-action-nextStateName1;` line.

Should be present `final:` line followed by all final states. The last final state has to be written followed by a dot (.) indicating the end of a file. If a final state is written in this line but it isn't present in transition line the document is refused.

Every time that the parser detect a mistake the system will show on screen the type of error and the line where this error appears.

The fields of this class are two string maintaining filename and typeError; a Boolean variable identifying ioError and an int with errorLine. Methods are:

```
static void setFileName(String fn);
```

```
static boolean checkFile(String filename, String content)
```

    this method checks file's correctness (file parser). Return true if document is correct, false otherwise and throws Exception during reading file.

```
static String readFile()
```

    this method reads a correct file with any extension supported. Return a string with file's content and throws Exception during the file's reading.

2009, 15th May

*Service*

New field of this class are:

- ✓ `HashMap<State, HashSet<Action>>` listAction: it stores information about all actions available from a state. The keyset represent all the service's states, instead elements are the list of actions feasible from a state;
- ✓ `HashMap<StateAction, HashSet<State>>` nextStates: it stores information on reachability of a state from a pair state-action. The keyset is all possible pair present state – action, the elements are list of states that we can reach from a present state performing an action;
- ✓ `Set<State>` finalStates: HashSet of final state;
- ✓ `Set<State>` notFinalStates: HashSet of not final states;
- ✓ `Set<TransitionFunction>` states: it stores all transition function;
- ✓ `String` filename;

Methods are:

- ✓ **Constructor**: `public Service(String readingFile)` that fills all the structures above;
- ✓ **Override**: `equals` and `toString`;
- ✓ **Other methods**: relative to get information on Service. These methods cost  $O(1)$  since all information are stored in HashMap structures ordered according appropriate key, i.e all action from a state are stored in listAction with key State so to find a determinate state the cost is one. These methods will always return information directly from fields of this class.

```
public Set<TransitionFunction> getTransitionFunctions()
```

get all the triples (present state, action and next state) stored from the document returning states field of this class;

```
public Iterator<State> getStatesWithoutFinal()
```

get all the states that aren't final state returning the iterator on not final state field;

```
public Iterator<State> getFinalStates()
```

get iterator on all final states of Service by final state field;

```
protected Set<State> getFinalStatesSet()
```

get all final states returning final state field. This method is protected because we don't allow manipulating a service's field in other package of the system. This because when a service is build it will never be modify from other classes;

```
public Iterator<State> getStates()
```

get all states of the service from keyset on listAction;

---

2009, 15th May

```
public boolean isFinalState(State state)
```

says if a state is final or not checking the isfinal property of the state;

```
public Iterator<State> getNextStates(State presentState, Action action)
```

get the next states from a present state and action returning an iterator on element list of nextStates that has presentState – action for key;

```
public boolean containsPresentAction(State presentState, Action action)
```

say if exist a pair of presentState and Action using contains method on keyset of nextStates field;

```
public Iterator<Action> getActions(State presentState)
```

get all possible action from a present state returning the element of a pair with presentState for key;

```
public String getName()
```

get the service's name;

```
protected void setName(String name)
```

set the service's name. This method is protected because we don't allow manipulating a service's field in other package of the system. This because when a service is build it will never be modify from other classes.

In this class we use to store information the HashMap class. This one organizes objects into pairs. There are two roles in each pair. These roles are *key* and *value*. A HashMap object organizes the pairs of objects so that no two pairs have equal keys. The internal data structure used to implement a HashMap allows an object pair to be found very quickly by looking for the object pair's key. The amount of time a HashMap takes to find an object pair by its key object is about the same for all object pairs in a HashMap and independent of the number of object pairs in the HashMap. So search, add, and remove operations are fast. The internal data structure used in a HashMap that allows a HashMap to quickly find object pairs by their key is called a *chained hash table*<sup>4</sup>.

---

4

Java Hashed Collections By Mark Gran

2009, 15th May

Our two HashMap structures are:

- ❖ listAction: this HashMap has for key an object of State class, that has two field a String and a boolean value, and for element an HashSet of Action's object, which have for field a String.

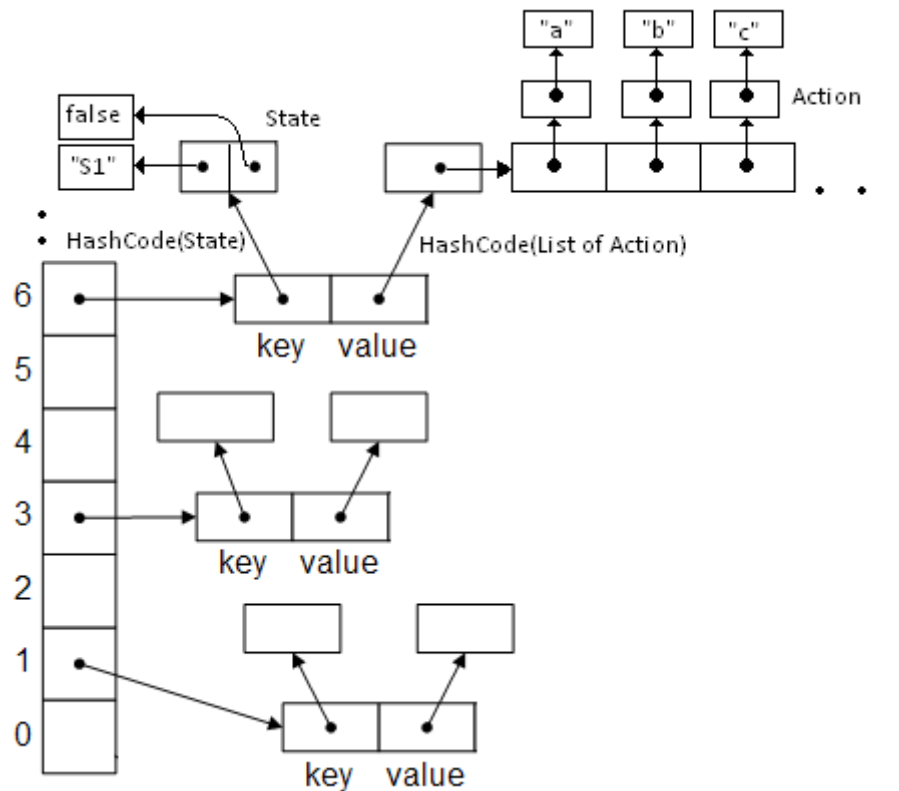


Figure 1: listAction structure



2009, 15th May

- ❖ **nextStates**: this HashMap has for key an object of StateAction class, that has two field a State and an Action, and for element an HashSet of State's object.

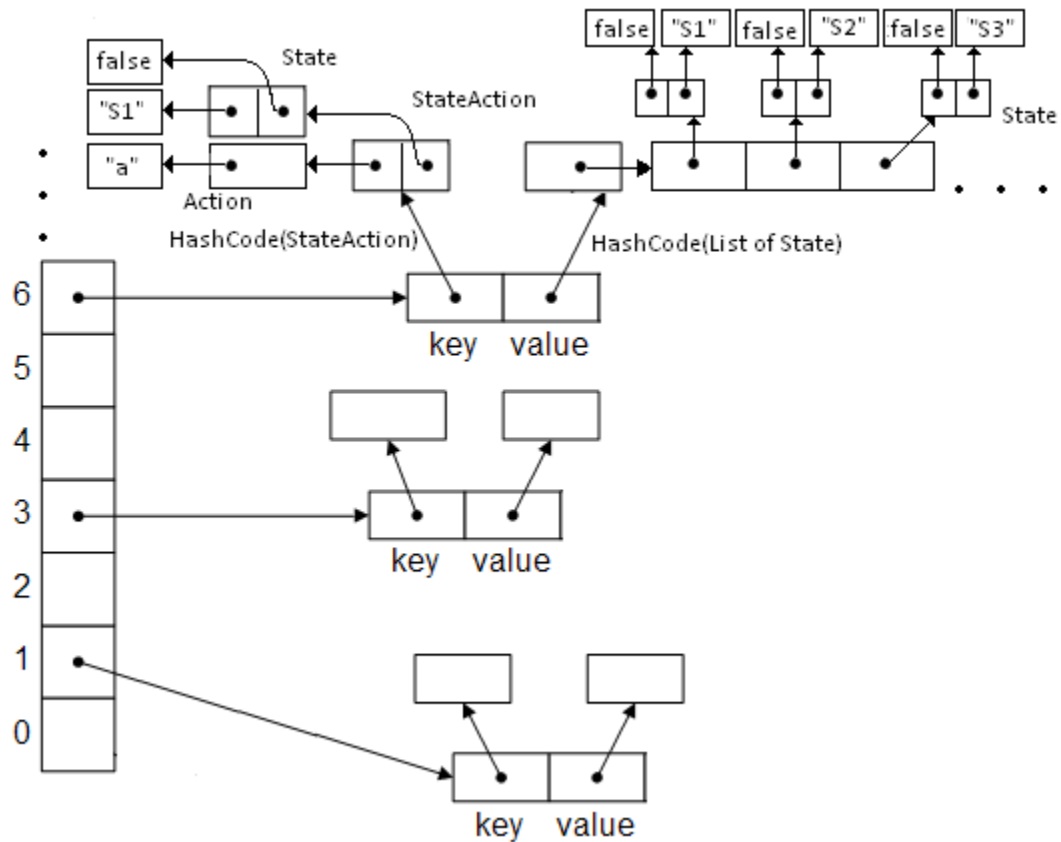


Figure 2: nextStates structure

### Community

This class stores all information about available services participating to community and all its characteristics. It implements Cloneable interface. Class fields are:

- ✓ Hashtable<Integer, Service> **community**: table with all available services. The keyset are position in Hashtable, the elements are services;
- ✓ HashSet<Integer> **emptyPosition**: list of all community's empty position that comes out from deleting services from community;
- ✓ int **maxk**: maintaining last key used in community;
- ✓ Hashtable<Integer, State> **CSRecord**: Hashtable of states for a single community state;
- ✓ Hashtable<Integer, Hashtable<Integer, State>> **Cstates**: Hashtable of community states;
- ✓ int **key**: key for Hashtable CStates;
- ✓ int **k** : key for Hashtable CFStates;
- ✓ Hashtable<Integer, State> **CFSRecord**;
- ✓ Hashtable<Integer, Hashtable<Integer, State>> **CFStates**;

2009, 15th May

Methods are:

- ✓ **Constructor:** empty constructor;
- ✓ **Override:** clone and toString;
- ✓ **Other methods:** are present get, add and delete methods to fill Hashtable community. Then we have a “contains method” that says if a service is present in community or not and a method that return the keyset on community. The other methods manage states, final or not, and records of community.

### Cartesian Product

Like in the previous version of this class we do the asynchronous product among available services. In this new version the principal method takes in input the community. To build the Cartesian product in this case we had empty the community to know when we can consider our computation finished, so we have done a clone of our community to avoid the problem of distorting this object.

Methods are:

- ✓ **executeProduct** : return a Cartesian product like Service. Service Class wants a String with content's available service. This string will be provide from productService method. The cost of this method is related, primarily, to the size of community, more precisely if community's size is equal to  $m$ , for each available service we have  $n$  state and  $a$  action, so the number of transition is  $N = n^2 * a$ , therefore the cost of this algorithm is

$$O(\text{community copy} + \text{productService first iteration} + \text{service cration} + (m - 2) * (\text{productService cost} + \text{service creation cost}))$$

The cost of product service depends on the number of service's transition. We give in input to productService method not a simple service, but a service that has always the number of transition that grows in an exponential way, i.e:

$$\begin{aligned} \text{First iteration:} & \quad |S_1| = N, |S_2| = N, \text{ cost is } O(N^2); \\ \text{Second iteration:} & \quad |S_1| = N^2, |S_2| = N, \text{ cost is } O(N^3); \\ \text{Third iteration:} & \quad |S_1| = N^3, |S_2| = N, \text{ cost is } O(N^4); \\ \dots & \\ \text{(m-2)-th iteration:} & \quad |S_1| = N^{m-1}, |S_2| = N, \text{ cost is } O(N^m); \end{aligned}$$

So total cost of this algorithm is:

$$O(m + N^2 + N^2 + (m - 2) * (N^m + N^m)) \leq O(2m * N^m)$$

2009, 15th May

```

public static Service executeProduct(Community comm){
    community = copy of community;
    String states = new String("");
    if(community.sizeCommunity()==0) return null;
    if(community.sizeCommunity()>2){
        Service service1=first community element;
        Service service2=second community element;
        //do the product between first two services and give it to a string
        states = productService(service1, service2);
        //create a new service with previous string
        Service service = new Service(states);
        Delete service1 and service2 from community
        while(community.hasMoreElements()){
            Service s3= third community service;
            //do product between s3 and previous product
            states= productService(service, s3);
            service = new Service(states);
            Delete s3 from community
        }
    }
    else{
        if(community.sizeCommunity()==2){
            do product between only two elements of community
        } else{ return the only service present }
    }
    return new Service(states);
}

```

- ✓ productService : This method built a string with Cartesian product of final states and transitions of two services. It uses productFinalStates and productTransition methods, so its cost is equal to

$$O(\text{producteFinalState} + \text{productTransition}) = O((n^2 * a)^2 + n^2) = O(N^2 + n^2) \leq O(N^2)$$

private static String productService(Service s1, Service s2)

- ✓ productTransaction : This method execute a Cartesian product among transaction functions. It does two cycle on the number of the service's transition. We assume that, in the worst case, the number of service's transition is equal to the number of service's state  $n$  multiplied by the number of action  $a$ , so are equal to  $N$ . This because the available services are non deterministic, therefore for each service is possible to think at the following scenario: every state can do every action available for the service. The cost of this method is equal to

$$O((n^2 * a)^2) = O(N^2)$$

2009, 15th May

```

private static String productTransition (Set<TransitionFunction> l1,
Set<TransitionFunction> l2){
    String transaction="";
    while(l1 is not finished){
        while(l2 is not finished){
            concat in transition four string with transition function text resulting
            from combination:
            ▪ l1 present state, l2 present state – l1 action – l1 next state, l2
            present state;
            ▪ l1 present state, l2 next state – l1 action – l1 next state, l2 next
            state;
            ▪ l1 present state, l2 present state – l2 action – l1 present state, l2
            next state;
            ▪ l1 next state, l2 present state – l2 action – l1 next state, l2 next
            state.
        }
    }
    return transaction;
}

```

- ✓ productFinalStates : This method execute a Cartesian product among final states. The cost of this algorithm is lower than the previous method, in fact the cycle is done on the number of final state, that, in general, is lower than the number of transition. At most, in the worst case, the number of final state is equal to the number of service's state  $n$ . So the cost of this algorithm is

$$O(n^2)$$

```

private static String productFinalStates (Set<State> l1, Set<State> l2){
    String prod = new String("");
    while(l1 is not finish){
        State final1= first state in l1;
        while(l2 is not finish){
            State final2 = first state in l2;
            prod=prod+final1+final2";";
        }
    }
    return prod;
}

```

In general Cartesian Product costs  $O(N^m)$ , that is EXPTIME in the number of services.

### 1.3 Third Version

Third version is the stable release in which we have introduced some arrangement about *side effect* (interference) and *computation*. First of all, we have insert only read iterator in our methods. In this way when a user uses a method, it returns an only read iterator on elements' set (see picture below). So remove operation is forbidden on this new iterator in opposite to reading operation. If the user tries to delete one element, application throws an exception. This behavior prevents "*interference*" on objects. We have decided to use this modality and not return a copy of this set because the second opportunity is more expensive.

```
/**
 * get all the states that aren't final state
 * @return Iterator on not final states
 */
public Iterator<State> getNotFinalState() {
    return new OnlyReadIterator<State>(notFinalStates.iterator());
}
/**
 * get all final states of Service
 * @return Iterator on final states
 */
public Iterator<State> getFinalStates() {
    return new OnlyReadIterator<State>(this.finalStates.iterator());
}
```

Figure 3: get Methods in Service class

We can see that these methods return *OnlyReadIterator* and not a structure's copy of *notFinalStates* and *finalState*. To understand better the real reasons of this choice, see the picture below that summarizes the situation:

	<i>Get's Methods</i>	<i>Advantages</i>	<i>Disadvantages</i>
<i>Without Shared Memory</i>	Return a deep copy (clone)	Client simpler	Waste memory
<i>With Shared Memory</i>	Return a <i>OnlyReadIterator</i>	Client trickier	Save memory

To have a better computation we have made a distinction between immutable and mutable object. Two objects are equal if they have the same memory location (same entity), this is the case of two object of *Service*, *State* and *Action* classes. An example of mutable objects is the *StateAction* class that manages the value of couple state and action; so our algorithms execute shallow and not deep equals. This is a way faster to compare two objects.

2009, 15th May

Before of this version there was a class `Community` used by both abstraction and control package, in fact this class had methods to manage available services and useful operations for simulation. To make application more modular as possible we have divided this class in two classes:

- ✓ `AvailableService` about all methods to manage available services. In this new class we modified available service's structure from `Hashtable<Service>` to `ArrayList<Service>` and related methods. This because, during simulation process, services are used in sequential way, so use a `HashSet` structure ordered according to a particular key is not so useful, in this case.
- ✓ `Community` uses `AvailableService` and methods to create Orchestrator.

Other modify has been introduced in the file to take in input. Until this version a textual file was:

```
transition:
S0-a-S1;
S1-b-S2;
final:
S2.
```

**Figure 4: textual file without initial state**

In this release an input file has an initial state because it is useful for Simulation so the file's formalism is the follow:

```
transition:
S0-a-S1;
S1-b-S2;
initial:
S0;
final:
S4.
```

**Figure 5: textual file with initial state**

Obviously to manage this situation has been modify file's parser, or rather `ReadFile` class, so when it reads a file doesn't throw an exception. Consequently the service's constructor has been modify to store a initial state in a new field `initial`.

2009, 15th May

To improve the usability of our program we have introduced some arrangements in Cartesian Product, in particular we added a label on the action of a service that represent a Cartesian product. Until previous version between two states there was only an action, while now there is action and service's name that can perform this action (picture below).

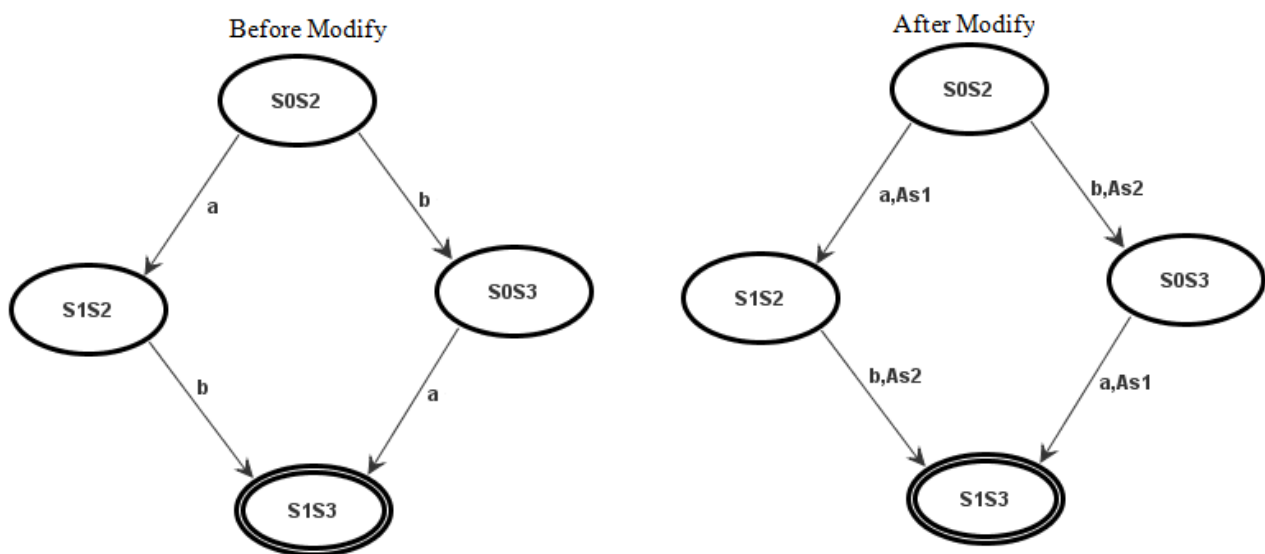


Figure 6: graph Cartesian product's service

The last change we have made for this package is the concept of deterministic or non-deterministic service. In this class we have introduced a new boolean field that gives information on this property. The reason why we have done this change is the necessity to know if the Target Service is deterministic, in fact if it isn't so the simulation is not executable.

## Chapter 2: Final release

In previous chapter we have described workflow of abstraction package (three version). Here we report the java classes' description of actual application.

### 2.1 Package description

The classes present in this package are:

- ✓ State;
- ✓ Action;
- ✓ StateAction;
- ✓ TransitionFunction;
- ✓ ConvertPdf;
- ✓ ReadFile;
- ✓ Service;
- ✓ Available Service;
- ✓ CartesianProduct.

#### *State*

This class represent a service's state. Its fields are a string that maintains the name of this state and a Boolean property isfinal. Methods are:

- ✓ **Constructor:** public State(String name) setting name in State class to the variable name and isfinal to false;
- ✓ **Override:** toString;
- ✓ **Other methods:** get on state name.

#### *Action*

This class represents a service's action. Its field is a string that maintains the name of this action. Methods are:

- ✓ **Constructor:** public Action(String name) setting action name;
- ✓ **Override:** toString;
- ✓ **Other methods:** get on action name.



2009, 15th May

These two classes represent immutable objects in fact there is only a “*method get*” to get action or state’s name and there aren’t methods to do side effect like add, set or remove. In Action and State classes we did not need to override equals method because they represent two object and not value.

### State Action

This is a support class used to maintaining information on pair state-action in service class. We are interest to know which states are linked by a specific pair of state-action. Its fields are a State presentState and an Action action. Methods are:

- ✓ **Constructor:** public StateAction(State presentState, Action action) setting presentState and action;
- ✓ **Override:** equals, hashCode and toString;
- ✓ **Other methods:** get on action and presentState;

This class represent an immutable value. Here we need to override equals method because it represent a value. Consequently we override hashCode Method.

### Transition Function

Fields of this class are: State presentState, Action action and State nextState. We override equals and hashCode method. This class is an immutable value representing triple of a transition system (presentState, action and nextState).

### Convert Pdf

Our Application can read a PDF file. To do this we created ConvertPdf class. It is used by ReadFile class to convert the content of pdf in textual one. In this class is used a special java library named **PDFBox-0.7.3.jar**. Methods are

```
protected static String GetTextFromPdf(String filename)
throws IOException, NoClassDefFoundError, Exception{
    String contentFile="";
    PDDocument document=null;
    InputStream in=new FileInputStream(filename);
    StringWriter out=new StringWriter();
    //use of pdf parser with document in input
    PDFParser parser = new PDFParser(in);
    parser.parse();
    //give to new document the parser one
    document = parser.getPDDocument();
    //creation of a text stripper to take document content
    PDFTextStripper stripper = new PDFTextStripper();
```

2009, 15th May

```

stripper.writeText(document,out);
contentFile=out.toString();
//close new document
document.close();
//return a string with file content
return contentFile; }

```

The catch exception has left to Read File class.

### Read File

Class used to make the parse of document in input. These documents could be:

- .txt;
- .aut;
- .doc, .docx;
- .pdf.

To be accepted by our application a file has to be written in this way:

```

transition:
S0-a-S1;
S1-b-S2;
initial:
S0;
final:
S4.

```

**Figure 1: textual file with initial state**

The first line of a document should be `transition:`. After this line the user writes all the transition of a service with the accuracy to do not put any blank space in `presentStateName1-action-nextStateName1;` line.

Should be present `initial:` line followed by a one state.

Should be present `final:` line followed by all final states. The last final state has to be written followed by a dot (.) indicating the end of a file. If a final state is written in this line but it isn't present in transition line the document is refused.

Every time that the parser detect a mistake the system will show on screen the type of error and the line where this error appears.

The fields of this class are two string maintaining filename and typeError; a Boolean variable identifying ioError and an int with errorLine. Methods are:

```
static void setFileName(String fn);
```

```
static boolean checkFile(String filename, String content)
```

this method checks file's correctness (file parser). Return true if document is correct, false otherwise and throws Exception during reading file.

2009, 15th May

```
static String readFile()
```

this method reads a correct file with any extension supported. Return a string with file's content and throws Exception during the file's reading.

### Service

This class represent a immutable object corresponding to a transition system. It has some fields

- ✓ `HashMap<State, HashSet<Action>>` listAction: it stores information about all actions available from a state. The keyset represent all the service's states, instead elements are the list of actions feasible from a state;
- ✓ `HashMap<StateAction, HashSet<State>>` nextStates: it stores information on reachability of a state from a pair state-action. The keyset is all possible pair present state – action, the elements are list of states that we can reach from a present state performing an action;
- ✓ `Set<State>` finalStates: HashSet of final state;
- ✓ `Set<State>` notFinalStates: HashSet of not final states;
- ✓ `State` initialState;
- ✓ `Set<TransitionFunction>` transitions: it stores all transition function;
- ✓ `String` filename;

In this class we use to store information the `HashMap` class. This one organizes objects into pairs. There are two roles in each pair. These roles are *key* and *value*. A `HashMap` object organizes the pairs of objects so that no two pairs have equal keys. The internal data structure used to implement a `HashMap` allows an object pair to be found very quickly by looking for the object pair's key. The amount of time a `HashMap` takes to find an object pair by its key object is about the same for all object pairs in a `HashMap` and independent of the number of object pairs in the `HashMap`. So search, add, and remove operations are fast. The internal data structure used in a `HashMap` that allows a `HashMap` to quickly find object pairs by their key is called a *chained hash table*<sup>5</sup>.

Methods are:

- ✓ **Constructor**: `public Service(String readingFile)` that fills all the structures above;
- ✓ **Override**: `toString`;
- ✓ **Other methods**: relative to get information on `Service`. These methods cost  $O(1)$  since all information are stored in `HashMap` structures ordered according appropriate key, i.e all action from a state are stored in `listAction` with key `State` so to find a

---

<sup>5</sup>

Java Hashed Collections By Mark Gran

---

2009, 15th May

determinate state the cost is one. These methods will always return information directly from fields of this class.

```
public Set<TransitionFunction> getTransitionFunctions()
```

get all the triples (present state, action and next state) stored from the document returning transitions field of this class;

```
public Iterator<State> getFinalStates()
```

get iterator on all final states of Service by final state field;

```
protected Set<State> getFinalStatesSet()
```

get all final states returning final state field. This method is protected because we don't allow manipulating a service's field in other package of the system. This because when a service is build it will never be modify from other classes;

```
public Iterator<State> getStates()
```

get all states of the service from keyset on listAction;

```
public boolean isFinalState(State state)
```

says if a state is final or not checking the isfinal property of the state;

```
public Iterator<State> getNextStates(State presentState, Action action)
```

get the next states from a present state and action returning an iterator on element list of nextStates that has presentState – action for key;

```
public boolean containsPresentAction(State presentState, Action action)
```

say if exist a pair of presentState and Action using contains method;

```
public Iterator<Action> getActions(State presentState)
```

get all possible action from a present state returning the element of a pair with presentState for key;

```
public State getInitialStates()
```

this method return a initial states of a service;

```
public boolean isDeterministic()
```

say if a service is deterministic or not;

```
public String getName()
```

get the service's name;

protected void setName(String name)

set the service's name. This method is protected because we don't allow manipulating a service's field in other package of the system. This because when a service is build it will never be modify from other classes.

These methods return a only read iterator. In this way when a user uses a method, it returns an only read iterator on elements' set (see picture below). So remove operation is forbidden on this new iterator in opposite to reading operation. If the user tries to delete one element, application throws an exception. This behavior prevents “*interference*” on objects. We have decided to use this modality and not return a copy of this set because the second opportunity is more expensive.

```
/**
 * get all the states that aren't final state
 * @return Iterator on not final states
 */
public Iterator<State> getNotFinalState() {
    return new OnlyReadIterator<State>(notFinalStates.iterator());
}
/**
 * get all final states of Service
 * @return Iterator on final states
 */
public Iterator<State> getFinalStates() {
    return new OnlyReadIterator<State>(this.finalStates.iterator());
}
```

Figure 2: only read iterator

### Available Services

In this class there is an *ArrayList<Service>* and related methods to manage all available services. This because, during simulation process, services are used in sequential way, so use a *HashSet* structure ordered according to a particular key is not so useful, in this case. This class will used by *Community* class.

Methods are:

- ✓ **Constructor:** empty constructor;
- ✓ **Override:** clone and toString;
- ✓ **Other methods:** are present get, add and delete methods to fill *ArrayList*. Then we have a “contains method” that says if a service is present in community or not and a method.

### Cartesian Product

This class does the asynchronous product among available services and it takes in input the available services object. To build the Cartesian product in this case we had empty the array List structure to know when we can consider our computation finished, so we have done a clone of our community to avoid the problem of distorting this object.

Methods are:

- ✓ *executeProduct* : return a Cartesian product like Service. Service Class wants a String with content's available service. This string will be provide from *productService* method. The cost of this method is related, primarily, to the size of community, more precisely if community's size is equal to  $m$ , for each available service we have  $n$  state and  $a$  action, so the number of transition, in worst case, is  $N = n^2 * a$ , therefore the cost of this algorithm is

$$O(\text{community copy} + \text{productService first iteration} + \text{service cration} + (m - 2) * (\text{productService cost} + \text{service creation cost}))$$

The cost of product service depends on the number of service's transition. We give in input to *productService* method not a simple service, but a service that has always the number of transition that grows in an exponential way, i.e:

First iteration:  $|S_1| = N, |S_2| = N$ , cost is  $O(N^2)$ ;  
 Second iteration:  $|S_1| = N^2, |S_2| = N$ , cost is  $O(N^3)$ ;  
 Third iteration:  $|S_1| = N^3, |S_2| = N$ , cost is  $O(N^4)$ ;  
 ...  
 (m-2)-th iteration:  $|S_1| = N^{m-1}, |S_2| = N$ , cost is  $O(N^m)$ ;

So total cost of this algorithm is:

$$O(m + N^2 + N^2 + (m - 2) * (N^m + N^m)) \leq O(2m * N^m)$$

```
public static Service executeProduct(Community comm){
    community = copy of community;
    String states = new String("");
    if(community.sizeCommunity()==0) return null;
    if(community.sizeCommunity()>2){
        Service service1=first community element;
        Service service2=second community element;
        //do the product between first two services and give it to a string
        states = productService(service1, service2);
        //create a new service with previous string
        Service service = new Service(states);
        Delete service1 and service2 from community
    }
}
```

2009, 15th May

```

        while(community.hasMoreElements()){
            Service s3= third community service;
            //do product between s3 and previous product
            states= productService(service, s3);
            service = new Service(states);
            Delete s3 from community
        }
    }

    else{
        if(community.sizeCommunity()==2){
            do product between only two elements of community
        } else{ return the only service present }
    }
    return new Service(states);
}

```

- ✓ productService : This method built a string with Cartesian product of final states and transitions of two services. It uses productFinalStates and productTransition methods, so its cost is equal to

$$O(\text{productFinalState} + \text{productTransition}) = O((n^2 * a)^2 + n^2) = O(N^2 + n^2) \leq O(N^2)$$

private static String productService(Service s1, Service s2)

- ✓ productTransaction : This method execute a Cartesian product among transaction functions. It does two cycle on the number of the service's transition. We assume that, in the worst case, the number of service's transition is equal to the number of service's state  $n$  multiplied by the number of action  $a$ , so are equal to  $N$ . This because the available services are non deterministic, therefore for each service is possible to think at the following scenario: every state can do every action available for the service. The cost of this method is equal to

$$O((n^2 * a)^2) = O(N^2)$$

private static String productTransition (Set<TransitionFunction> l1,

Set<TransitionFunction> l2){

String transaction="";

while(l1 is not finished){

while(l2 is not finished)){

concat in transition four string with transition function text resulting from combination:

- l1 present state, l2 present state – l1 action – l1 next state, l2 present state;

2009, 15th May

```

        ■ l1 present state, l2 next state – l1 action – l1 next state, l2 next state;
        ■ l1 present state, l2 present state – l2 action – l1 present state, l2 next state;
        ■ l1 next state, l2 present state – l2 action – l1 next state, l2 next state.

    }
}
return transaction;
}

```

- ✓ **productFinalStates** : This method execute a Cartesian product among final states. The cost of this algorithm is lower than the previous method, in fact the cycle is done on the number of final state, that, in general, is lower than the number of transition. At most, in the worst case, the number of final state is equal to the number of service's state  $n$ . So the cost of this algorithm is

$O(n^2)$

```

private static String productFinalStates (Set<State> l1, Set<State> l2){
    String prod = new String("");
    while(l1 is not finish){
        State final1= first state in l1;
        while(l2 is not finish){
            State final2 = first state in l2;
            prod=prod+final1+final2"";
        }
    }
    return prod;
}

```

In general Cartesian Product costs  $O(N^m)$ , that is EXPTIME in the number of services.



## 2.2 Implementation Choices

In this paragraph we want to discuss about some implementation choices: Cartesian product and a Service's constructor.

Service's constructor takes a string in input because:

- ✓ It permits us to make independent building of a service from reading a file, in fact read a file and build a service are two distinct operations; the reason of this idea is that: we can decide to change how to read a file, i.e. from an url or web page, but service class is always the same.  
The important is that the string respects our standard.
- ✓ it gives us the possibility to manage better a transition system, in fact, our formalism on a correct string gives the possibility to distinguish among different categories of information on a service, like transitions, final states or initial state

These are the benefits to have a service's constructor. However, there is a disadvantage not so relevant: to build Cartesian product among service we have to create a string with all the information on the service that represent our product and give it to service's constructor.

Usually, the Cartesian product it's costly, from computational point of view, but our implementation is more costlier than natural, because we do the follow steps:

- ✓ we do a Cartesian product among transition function of two services (presentState-action-nextState) and we return a string;
- ✓ we do a Cartesian product among final states of two services and we return a string;
- ✓ we do a Cartesian product among initial states of two services and we return a string;
- ✓ we build a string with the three previous strings ;
- ✓ we give this string to a service's Constructor and we create a new service representing a Cartesian Product between these two services.
- ✓ Repeat this cycle between this new service and another service and so on for all services

It's true that this product it's very costly but this one does not influence simulation and orchestrator algorithms because they use a Cartesian product built "*on the fly*". This product is something more offers by our application to users so they can decide if execute it or not.

---

2009, 15th May

To reduce this cost it is possible to do some arrangements. The general idea to follow should be implements these steps in a new Cartesian product :

- ✓ Do a new class that extends Service and use it in a different way respect to actual Service class;
- ✓ During the Cartesian product between two service maintain the information on this product "*on the fly*", without storing it using this new class;
- ✓ Once we obtained the last product, resulting from previous steps, we stores it like a Service of actual class.

## **Second Part:**

## **Control Package**

## Overview

An orchestrator is a function which selects an available service to execute the action requested by the (target-conformant) client.

In order to be able to perform this function we have to create an Orchestrator Generator (OG). Orchestrator Generator is a finite state machine that, given a target service and a set of available services, can say (with a function  $\omega$ ) which are the services among available services performing a given action, according to the maximal simulation relation. And for each choice of one of such services it progresses to the next state.

Once we have OG, we get orchestrators by simply picking up, at each step, one among the services returned by  $\omega$  and in this way we can be sure that the selected service is able to execute the assigned action.<sup>6</sup>

Building an orchestrator we want to check the existence of service composition by checking the existence of a simulation relation between target and community, which is a transition system representing the asynchronous product of all available services.

According to the definition of simulation, given two transition systems  $TS_t$  (which represent a target service) and  $TS_C$  (for the community), a simulation relation of  $TS_t$  by  $TS_C$  is a relation  $R \subseteq S_t \times S_C$ , such that:

$R(s_t, s_C)$  implies:

1. if  $s_t \in F_t$  then  $s_C \in F_C$ ;
2. for all transitions  $s_t \xrightarrow{a} s'_t$  in  $TS_t$  there exists a transition  $s_C \xrightarrow{a} s'_C$  in  $TS_C$  and  $R(s'_t, s'_C)$ .

where  $s_t$  is a state of target,  $s_C$  is a state of community,  $F_t$  and  $F_C$  are final states of target and community respectively, and  $a$  is an action.

<sup>6</sup>

*Automatic Service Composition Via Simulation – Daniela Belardi, Fahima Cheikh, Giuseppe De Giacomo, Fabio Patrizi*

The definition says that state  $s_t$  of  $TS_t$  is in a simulation relation  $R$  with  $s_C$  of  $TS_C$  if:

- if  $s_t$  is final then also  $s_C$  is final;
- for every action  $a$  and state  $s'_t$ , if  $s_t$  can make a transition to  $s'_t$  with action  $a$ , then also  $s_C$  can make a transition to some  $s'_C$  with action  $a$ , in such a way that  $s'_t$  is still in the same simulation relation  $R$  with  $s'_C$ .

A composition of services exists if  $R$  is a simulation relation of  $TS_t$  by  $TS_C$  such that  $R(s_{0t}, s_{0C})$ , i.e. if the initial states of target and community are in the set of simulation. This is the so called maximal simulation relation and in this case we can say that  $TS_t$  is simulated by  $TS_C$ .<sup>1</sup>

In this project we want to realize an orchestrator finding a service composition which corresponds to find a simulation relation between two transition systems: target and community.

Therefore in the realization of this program we have to implement four main steps:

1. build the community
2. calculate the simulation set
3. check the existence of composition
4. provide a service selection function  $\omega_r$ .

## Chapter 1: Control Package Description

In this package there are all java classes which contain the main algorithms to execute orchestrator, such as simulation.

The package is composed by these classes:

- ✓ *CState*: it represents a state of the community given by the Cartesian product of states of services in the community.
- ✓ *Community*: it should be a transition system which is the result of the asynchronous product of all available services. In our implementation it contains only the states of this new transition system, and can return also final and initial states of the community (of *CState* type).
- ✓ *SimulatedBy*: it represents a pair of states which could be in simulation. So it is made by a state of the target service and a state of the community.
- ✓ *Simulation*: this class contains methods to create a simulation set given a target service and a community.
- ✓ *OrchestratorKey*: it is formed by an object *SimulatedBy* and an action. It allows to store a triple with a target state, a community state and an action as a key for a table that will be the result of orchestrator.
- ✓ *Orchestrator*: it is the responsible class to build an orchestrator given a target service and a community. The result of composition is stored in a table with *OrchestratorKey* objects as keys and services as values.

### 1.1 Simulation algorithm

The core of this system is given by the implementation of the algorithm of simulation. In fact if we have the output of this algorithm we can say if a composition exists and, in positive case, we can build orchestrator generator.

As we have explained above, to check simulation we need a community, which is a new transition system given by the asynchronous product. In our realization this product is implemented in the package abstraction with the CartesianProduct class. In theory we have to pass an instance of it to our simulation algorithm but, during the analysis phase, we have thought that in this way the algorithm can be much expensive, because this structure can be very big and when we have to check all the simulation relations we have to extract information we need from this complex structure. Moreover we have observed that in each step of simulation we should take a state of community and from this one we have to check if there are the same actions of target state and, if they are, we should verify also that next states of community and target service are in simulation. Since community's state is a tuple like  $\langle s_0, s_1, s_2, s_3 \rangle$  where  $s_i$  is a state of the  $i$ -th service and because we know the services we can obtain directly from services which are the states in a community state having a certain action and we can build "on fly" the next state of Community. For instance, if we have a target's state with action "a", a community's state  $\langle s_0, s_1, s_2 \rangle$  and we know that the Service related to state  $s_0$  can perform action "a" going to the next state  $s_0'$ , then we can say that in the community also there is an edge with action "a" going from state  $\langle s_0, s_1, s_2 \rangle$  to  $\langle s_0', s_1, s_2 \rangle$ . Using this strategy we can avoid the use of the complete Cartesian Product of Services.

According to this idea we need a method that, given a set of available services, allows to create all the states of the community.

Once we get these states we can apply a simulation algorithm like this:

```
void simulation (Service target, Community community) {
    Set R = createSimulationSet(target, community);
    1:   while (!R.isEmpty()){
        for each line r in R do{
            State ts = r.getTargetState();
            CState cs = r.getCommunityState ();
        2:   for each action a in ts.getActions(){
            3:   for each state s in cs {
                    if(service.containsAction(a, s)){
                        Set next = s.getNext(a, s);
                        for each state s' in next {
                            CState cstate = buildRecord (cs, s', i);
                            newRecord = <ts.getNextState(), cstate>;
                            if(!R.contains (newR)){
                                continue to 3;
                            }
                        }
                    }
                }
            continue to 2
        }
    }
}
```

2009, 15th May

```

        R.remove(r);
        continue to 1;
    } //end for each action
} //end for each line
exit;
} //end while
}

```

First instruction of this algorithm:

Set  $R = \text{createSimulationSet}(\text{target}, \text{community})$ ;

it allows to get an initial set of simulation where there are pairs like  $\langle t_0, x_0y_0z_0 \rangle$  where  $t_0$  is a state of the target and  $x_0y_0z_0$  is a state of the community. The `createSimulationSet` method is responsible to create the initial set inserting in it each possible pair formed by:

- a target's final state and a community's final state
- a not target's final state and a community state (final and not final).

In this way we assure the satisfiability of the first condition of simulation.

After that we have to check the second condition of simulation: it is done picking each pair of states from just created simulation set and checking, for each of them (for example for  $\langle t_0, x_0y_0z_0 \rangle$ ), if the target state ( $t_0$ ) is simulated by the community state ( $x_0y_0z_0$ ).

For each pair like  $\langle t_0, x_0y_0z_0 \rangle$  we can have several cases:

- ✓ target state ( $t_0$ ) has an action that cannot be performed by anyone of available services when they are in the state specified in this community state ( $x_0y_0z_0$ ): in this case the pair is deleted from simulation set and we have to start checking simulation again from the first pair in the set.
- ✓ target state ( $t_0$ ) has an action that can be performed by some services (if they are in the state specified in this community state ( $x_0y_0z_0$ )) but every next community state (one for each next state of services which can do the action) isn't in the simulation set with next target state: in this case the pair is deleted from simulation set and we have to start checking simulation again from the first pair in the set.
- ✓ target state ( $t_0$ ) has all the actions that can be performed by some services (if they are in the state specified in this community state ( $x_0y_0z_0$ )), and next states are simulated by someone next community state: in this case the pair remains in the simulation set and we should continue our control on another pair.

So, when there is a pair which is removed from the simulation set the check must begin again from the first record in this set, otherwise the checking is done over each pair of the state. After all pairs will be verified we can stop checking and the result set will be our simulation set.

Once we get this set we should verify if composition exists: for this reason we need a method that checks if initial state of target service is simulated by initial state of the community, that is true if target's initial state and community's initial state compose a record that is present in the simulation set.

## 1.2 Simulation Implementation

We have chosen to create a class `Simulation` where there are some methods permitting to execute every steps of simulation. But this class has a protected constructor because we want that the execution of this algorithm is dependent from the execution of orchestrator. In fact we decided to create simulation when an instance of orchestrator is created.

`Simulation` class contains this relevant method:

```
protected void compositionViaSimulation () {  
    this.createSimulationSet();  
    boolean modified = true;  
    while (!pairs.isEmpty() && modified) {  
        modified = checkSimulationSet();  
    }  
}
```

**Figure 1: compositionViaSimulation method**

This is the main method to create simulation set. Here there is, first of all, an invocation to the `createSimulationSet()` method that is responsible to create initial set of simulation satisfying the first condition. After that a boolean variable is used permitting to understand when all simulation set's records will be checked. So `checkSimulationSet()` method will be called every time this variable has true as value, i.e. each time `checkSimulationSet()` will return true.

The worst case of iteration number is when all records into simulation set must be deleted. In this case `checkSimulationSet()` is called a number of times equals to number of records into simulation set. Given the set of target states  $S_t$ , the set of final target states  $F_t$ , the set of states of each available service  $S_i$  and the set of final states for them  $F_i$ , we can say that simulation set should contain this set of records after calling `createSimulationSet()`:



$$((S_t - F_t) \times (S_1 \times S_2 \times \dots \times S_m)) \cup (F_t \times (F_1 \times F_2 \times \dots \times F_m)) \quad (1)$$

If  $F_t = S_t$  records are only:

$$(F_t \times (F_1 \times F_2 \times \dots \times F_m))$$

and if  $|F_i| < |S_i|$  for some or all Services then we get the lower number of records of initial simulation set and then the lower number of iterations for this loop.

Instead if  $|F_i| = |S_i|$  for each available service  $i$  then number of records is equal to elements of:

$$((S_t - F_t) \times (S_1 \times S_2 \times \dots \times S_m))$$

In general, we have that  $|F_i| < |S_i|$ , also for the target service, so we have a set as (1) and in this case the number of records will be:

$$((|F_t| - |S_t|) * (|S_1| * |S_2| * \dots * |S_m|)) + (|F_t| * (|F_1| * |F_2| * \dots * |F_m|)) \quad (2)$$

If we assume that  $|S_t| = x$ ,  $|F_t| = y$  and  $|S_i|_{max} = n$ ,  $|F_i|_{max} = p$ , we can say that:

$$(2) \leq (x - y) * n^m + y * p^m \leq x * n^m \text{ (case for } p = n)$$

Hence, in worst case number of iteration of this loop will be  $x * n^m$ . (3)

```
private void createSimulationSet () {
    Iterator<State> ts = this.target.getStates();
    while (ts.hasNext()) {
        State t = (State)ts.next();
        if(this.target.isFinalState(t)) {
            Iterator<ArrayList<State>> fcs = this.community.getFinalCommunityStates();
            while(fcs.hasNext()) {
                ArrayList<State> s = (ArrayList<State>)fcs.next();
                SimulatedBy newRecord = new SimulatedBy(t, s);
                this.pairs.add(newRecord);
            }
        }
        else {
            Iterator<ArrayList<State>> cs = this.community.getCommunityStates();
            while(cs.hasNext()) {
                ArrayList<State> s1 = (ArrayList<State>)cs.next();
                SimulatedBy newRecord = new SimulatedBy(t, s1);
                this.pairs.add(newRecord);
            }
        }
    }
}
```

Figure 2: createSimulationSet method

This method allows to create initial simulation set satisfying first simulation's condition. For that it takes all states of target and for each of them it controls if it is final one: if it is final it creates

records with this target state and every final community state; if it isn't final it inserts new records with this target state and each one of community states.

Cost of this method is equal to the cost to build community states and final community states one time (because getCommunityState method creates these states only at first invocation) plus the cost to create new records and to add them to the simulation set that is an HashSet and so this operation requires a constant time. Therefore if  $x$  is the number of states of target service and  $n^m$  the cost of community.getCommunityStates(), total cost for this method is:

$$2n^m + x * n^m = O(x * n^m) \quad (4)$$

```
private boolean checkSimulationSet() {

    Iterator<SimulatedBy> record = pairs.iterator();
    while (record.hasNext()) {
        SimulatedBy line = record.next();
        State ts = line.getTargetState();
        CState cs = line.getCommunityState();
        Iterator<Action> action = this.target.getActions(ts);
        while(action.hasNext()) {
            Action as = action.next();
            Iterator<State> nextState = this.target.getNextStates(ts, as);
            while (nextState.hasNext()) {
                State nextTState = (State)nextState.next();
                if (!checkSimulationOfNextState(cs, as, nextTState)) {
                    this.pairs.remove (line);
                    return true;
                }
            }
        }
    }
    return false;
}
```

**Figure 3: checkSimulationSet method**

Here there is the main part of simulation algorithm. It returns true if one record has been removed, false only if every records of simulation set have been checked without deleting anything (in this case the simulation algorithm will be stopped).

Every time this method is called it gets records from simulation's HashSet one by one and for each of them it takes the stored target state and the community state. Then, it gets all possible actions from that target state and, for each of them, the next target state. So it is able to call checkSimulationOfNextState(cs, as, nextTState) method that will control the second simulation condition: if there is one action for that checkSimulationOfNextState returns false then this record can be removed from simulation set, stopping test on this pair of states; otherwise it means that this pair satisfies simulation condition.

The most external loop will be executed, as many times as number of records of simulation set until there is one of them that must be deleted. Thence worst case of execution of this method will be when the first record to delete will be in the last position of the set. In this case number of iteration will be equal to the number of records. In each iteration the cost to get target state and community state from a record is constant; then it gets all actions of a target state (also this one is constant) and for each action it takes next state and it invokes `checkSimulationOfNextState(...)`. In the case it returns false record will be removed from HashSet with constant time. Therefore total cost of this method will be:

$$x * n^m * (b * O(\text{checkSimulationOfNextState})) \quad (5)$$

Where  $x * n^m$  is the number of records in the worst case,  $b$  is the maximum number of actions which a target state gets, and cost is the sum of constant time operations.

```
private boolean checkSimulationOfNextState (CState cs, Action as, State nextTState)
    for (int i = 0; i < cs.getSize(); i++){
        Service serv = this.community.getService(i);

        if (serv.containsPresentAction(cs.get(i), as)) {
            boolean candoit = true;
            Iterator<State> nextState = serv.getNextStates(cs.get(i), as);
            while (nextState.hasNext() && candoit) {
                State next = (State)nextState.next();
                CState hs = cs.buildNewState (i, next );
                SimulatedBy rec = new SimulatedBy (nextTState, hs);
                if (!this.contains(rec)) {
                    candoit = false;
                }
            }
            if(candoit) return true;
        }
    }
    return false;
}
```

Figure 4: `checkSimulationOfNextState` method

With this method we check if one target state with a specified action and a next state is simulated by a given community state. Thus it controls each state of available services contained in that community state to find someone that can simulate the target state. For each state (in community state) it gets available service to which it belongs. In this way we are able to check if that state has the given action: if it has we can gain also what are next states for that action and, for each of them, build a new record composed by next target state and next community state (obtained substituting in the given community state one of the found next states in the position relating to that of the service which has that action and that next state). When each one of this record is created and we have checked that they are in the simulation set, it means that there is one community state that can simulate the target state for the given action and this test can be stopped here with positive result. Instead if for every state (in the community) we haven't a next community state that simulates the target state for that action then test has a negative result.

In worst case this method should analyze every state in a community state and if the number of services is  $m$  the dimension of a community state is  $m$ . Then, for each of them it should take the related service (with constant cost) and check if it contains that action performing on a given state (constant time). If so, it should take all next states reachable from that state with the given action and for each of them (until one will satisfy the simulation condition) it should build the next community state (in a time equals to that to get a copy of the present community state that is  $O(m)$ ) and checking if it is present in the simulation set (constant time). Obviously in the worst case total cost will be:

$$m * (n * m) = O(m^2 * n) \quad (6)$$

Ultimately, putting together (4) (5) (6) (7), cost to execute this simulation algorithm is equal to:

$$\begin{aligned} x * n^m + x * n^m * (x * n^m * (b * m^2 * n)) &= (x * n^m) + (x^2 * n^{2m+1} * (b * m^2)) \\ &\leq (n^{(m+1)} + n^{(2m+3)} * b * m^2) = O(n^{2m} * b * m^2) \end{aligned} \quad (7)$$

The result (7) is obtained assuming that the number of states of target is less or equal to that of an available service, i.e.  $x=n$ .

Here we can see that, a part from multiplicative factors, cost of our simulation algorithm is exponential in the number of services; so it shows that this algorithm is not worse that the cost of a composition via simulation.

### 1.3 Orchestrator class

As we said before, we chose to start simulation algorithm contextually to the creation of an orchestrator instance. In this way we are sure that the result of one orchestrator is connected only to the right simulation result.

In fact the constructor of Orchestrator Class invokes the method `compositionViaSimulation()` of class `Simulation` creating the simulation set.

In this class there are also two main methods: one to check if composition exists and another to build an orchestrator.

The first one is:

```
public boolean checkComposition () {

    SimulatedBy initPair = new SimulatedBy (this.target.getInitialState(),
                                              this.community.getInitialStates());

    return this.s.contains(initPair);
}
```

Figure 5: checkComposition method

---

2009, 15th May

Here one pair of states is built with the initial state of target service and initial state of community. Then it controls that this pair is present in the simulation set (if it is contained then composition exists, otherwise it doesn't exist).

Cost for this method is constant and equal to that to obtain initial state of community:  $O(m)$  if  $m$  is the number of available services. (cost of `contains(initPair)` is constant).

The second method is:

2009, 15th May

```

public boolean generateOrchestrator () {
    if (this.s.isEmpty())
        return false;
    else {
        this.orchestrator = new HashMap<OrchestratorKey, Set<Service>> ();
        Iterator<SimulatedBy> pairs = s.getSimulationSet();
        while (pairs.hasNext()) {
            SimulatedBy record = pairs.next();
            State ts = record.getTargetState();
            CState cs = record.getCommunityState();
            Iterator<Action> action = this.target.getActions(ts);
            while (action.hasNext()) {
                Action as = action.next();
                Set<Service> services = new HashSet<Service>();
                Iterator<State> nextTS = this.target.getNextStates(ts, as);
                State nextTState = (State)nextTS.next();
                for (int i = 0; i < cs.getSize(); i++){
                    Service serv = this.community.getService(i);
                    if (serv.containsPresentAction(cs.get(i), as)) {
                        boolean candoit = true;
                        Iterator<State> nextState = serv.getNextStates(cs.get(i), as);
                        while (nextState.hasNext() && candoit) {
                            State next = (State)nextState.next();
                            CState hs = cs.buildNewState (i, next );
                            SimulatedBy rec = new SimulatedBy (nextTState, hs);
                            if (!s.contains(rec))
                                candoit = false;
                        }
                        if(candoit && !contains(services,serv))
                            services.add(serv);
                    }
                }
                OrchestratorKey key = new OrchestratorKey (record, as);
                orchestrator.put(key, services);
            }
        }
    }
    return true;
}

```

Figure 6: generateOrchestrator method

This method allows to get a map where for each tuple  $\langle t_i, s_1 s_2 s_3 \dots s_m, a \rangle$ , formed by a target state, a community state and an action, it can say which are these among available services that are able to simulate the action  $\langle a \rangle$  for the target state  $\langle t_i \rangle$  starting from the community state  $\langle s_1 s_2 s_3 \dots s_m \rangle$ .

This map can be created only if the simulation set is not empty. If this condition is verified this method gets each record from the simulation set and for each one it gets the target state and the community state; for each action of a target state it controls which are the available services able to simulate it and adds them to a new set relating to this target state, this community state and this action. Once all the services that can do that action are added, a new record is stored in the orchestrator map with these values.

So complexity of this method is given, first of all, by cost of the most external loop:

```
while (pairs.hasNext())
```

This loop is run a number of time equals to the number of records stored in the simulation set: this number is lower or equals to  $x * n^m$ , where  $x$  is number of target's states,  $n$  is the number of community's states and  $m$  is the number of services.

At each iteration there is a loop getting each action for a target state:

```
while(action.hasNext())
```

This loop is executed (for a target state) as many times as the number of possible actions for the considered target state. We assume it is  $b$ .

For each action of a target state we have to check, for each state of available services contained in the community state, which are the available service states able to execute that action:

```
for (int i = 0; i < cs.getSize(); i++)
```

It means one iteration for each available services and we assume they are  $m$ .

At last we have this other loop:

```
while (nextState.hasNext())
```

controlling, for each available service's state able to perform the considered action, if next community state simulates the target next state for a given action. If we assume that a service can have the searched action going to a number of next states, that in the worst case is  $n$ , and that building a new record to check cost  $m$  (the number of services), we can say that cost of this loop is  $n * m^2$ .

So the total cost of this method is:

$$x * n^m * b * m^2 * n = O(x * n^{(m+1)} * b * m^2)) \quad (8)$$

If we assume that the number of target states  $x$  is less than the maximum number of states of available services we can say:

$$(8) \leq O(n^{(m+2)} * b * m^2))$$

This means that also the cost of orchestrator is exponential in the number of services.

Another relevant method in the class Orchestrator is:

```
public Iterator<Service> getServicesForStateAction(OrchestratorKey)
```

It allows to get all available services which can execute a given action starting from a specified target state and a community state remaining in simulation.

This is done simply with a call to the method `get()` of `HashMap` that, given a key (in this case an object `OrchestratorKey`), returns the value (a Set of Services). So the cost for this operation is  $O(1)$ .

2009, 15th May

These described methods have been used to create an orchestrator generator that is a graph (that we represent with a Service object), with each state composed by a target state and a community state that simulate it, presenting all the states reachable from initial states and acting all actions of target states.

The creation of such orchestrator generator is possible using the method `createOrchestratorGenerator()` which return a service, building through this algorithm:

```

createOrchestratorGenerator() {
    if composition exists {

        Service orchestratorGenerator = ∅;
        Set<SimulatedBy> finalStates = ∅;
        Set<SimulatedBy> toAdd = ∅;
        Set<SimulatedBy> alreadyAdded = ∅;
        State initialTS = target.initialState;
        CState initialCS = community.initialState;
        SimulatedBy initialStates = <initialTS, initialCS>;
        toAdd.add(initialStates);

        for each pair sim in toAdd {
            alreadyAdded.add(sim);
            State initTS = sim.targetState;
            CState initCS = sim.communityState;

            if (initTS.isFinal & initCS.isFinal)
                finalStates.add(sim);

            for each action a of initTS {
                State nextTS = target.getNextStates(initTS, a);
                OrchestratorKey key = <sim, a>;
                Set<Service> serv = orchestrator[key];
                for each service si in serv {
                    Set<State> nextCommunityState =
                        si.getNextStates(initialCS[i], a);
                    for each state ns in nextCommunityState {
                        CState cs = initialCS.copy;
                        cs[i] = ns;
                        SimulatedBy newSim = <nextTS, cs>;
                        orchestratorGenerator.add(<initialStates-a-
                            newSim>;);
                        if (!alreadyAdded.contains(newSim))
                            toAdd.add(newSim);
                    }
                }
            }
        }
    }
}

```



2009, 15th May

```
        toAdd.remove(simulated);
    }
    orchestratorGenerator.add(<initialStates>);
    for each finalSim in finalStates
        orchestratorGenerator.add(<finalSim>);
    }
}
```

Obviously, before building an orchestrator generator (OG) we have to check if composition exists: if doesn't exist we cannot create an OG. If it exists we have to build a new service starting from the initial states of target and community, adding a new transition relation in the service for this pair of states (that represent a state of OG) and each possible action for that target state. Then, for every other pair of states present in the orchestrator map and reachable from the initial states we have to add other transition relations like for the initial states. When there aren't other states reachable we should add only initial and final states and so the service OG will be create.

#### 1.4 Implementation's choices

During the implementation we had to improve our code Control package and here we summarize the main choices we took and the reasons that led us to do some modifications:

- ✓ Initially we had a Community class containing methods to create community states but also to add and to delete available services. Then we have thought to separate the part regarding the management of available services (realizing an AvailableServices class in abstraction package) from that relative to the creation of community's states. This was done to be able to create a set of community states without caring about the possible modifications on the services set. So community states can be created just one time for each set of available services (don't need to be created each time).
- ✓ At the beginning CState class contained a state of the community in the type ArrayList. But changing the class Community (as we said above) we can know what is the dimension of a community's state (it is given by the number of available services passed to the Community constructor). So we replaced ArrayList with a static Array structure. In fact we tested the system with both the two structure so we could appreciate the difference between run times. In particular we tested the system with four available services, each one with ten states, and a target service with five states: with ArrayList time to end the execution of simulation was about 15-16 minutes; with array it takes only 8-9 minutes. So we preferred to use array.

## Chapter 2: Creation of community states “on fly”

In this chapter we want to explain the reasons which led us to make some modification to the initial structure of control package. The idea is that when we build community states with methods of Community class we have to create and store all states of a community. This means that our plan to compute a community “on fly” is not realized completely because in the first step of simulation, i.e. when we create the initial simulation set (with createSimulationSet() method of Simulation class), we build all community states.

Practically what we wanted to obtain was that each time we need a state of a community we create it keeping track of each state that is already returned, so in the next time a state is needed a different state is returned. In this way we can have all the states of the community without storing all of them in memory but as if they were already built (in a virtual way).

This means that realizing some methods able to perform this mechanism we can save much used memory's space. Instead in terms of mid time of execution our algorithms should have the same value and they will be discussed in the following pages.

### 2.1 Control Package v2

Control package was modified in order to implement the idea discussed above of a “virtual community”.

In this way this package contains the same classes discussed in the chapter 1 but also other two classes: *IteratorCommunityStates* and *IteratorFinalCommunityStates*.

These classes implement the interface *Iterator* and they are used to return a state of community and pointing to another state.

Hence classes of this package in the version 2 are:

- ✓ *CState*: representing a state of the community.
- ✓ *Community*: it contains methods to return an iterator on all states and on final states of community and to get initial state of a community.
- ✓ *IteratorCommunityStates*: it represents an iterator on a set of community states.
- ✓ *IteratorFinalCommunityStates*: it represents an iterator on a set of final community states.
- ✓ *SimulatedBy*: it represents a pair of states which could be in simulation.
- ✓ *Simulation*: this class contains methods to create a simulation set given a target service and a community.
- ✓ *OrchestratorKey*: it is formed by an object *SimulatedBy* and an action.
- ✓ *Orchestrator*: it is the responsible class to build an orchestrator given a target service and a community.

2009, 15th May

The only class that is changed in this version is the class Community. In fact in the previous version this class had as fields two set of states which stored all community states and all final states. In this new implementation they are not necessary because we don't need to store states but we have some methods returning only an iterator on a community state. This iterator is in the type IteratorCommunityStates for all community state and IteratorFinalCommunityStates for final community state. So, while in the previous version getCommunity() method builds and stores all the set of community states and return an iterator on this set, in the second version the same method creates only a single state and return an iterator on this element.

Therefore in terms of use of space we can say that the old implementation is much more expensive than the new one: first implementation of Community class needs a space equals to that to store all states of a community; second implementation needs only space for a single community state.

Instead, regarding the execution time of getCommunityState() we can say that:

- for the first version it is equal to time to creates all states so  $O(n^m)$ , if  $n$  is the number of states of an available service and  $m$  is the number of available services
- for the second version it equals to time to create a state, then it is  $O(m)$  if  $m$  is the number of available services

It could seem that with this modification the algorithm of simulation is quicker than it was in the first version. But, because the createSimulationSet() method of Simulation class gets all states of community for each target state, with the new version we have to create all states of community for each target state, so the cost for this step is  $O(x \cdot n^m)$  if  $x$  is the number of states of a target service and  $n$  and  $m$  are defined as above. We can see that it was also the cost we got in the previous version (see chapter 1.2) , even if, in that case, the creation of all states was done only at the first called of that method.

In conclusion, we can say that using this structure we reduce the memory space but not the time of execution

---

2009, 15th May

# **Third Part:**

## **Presentation Package**

## Chapter 1: Graphical user interface

The Swing classes implement a set of components to build graphical user interfaces and adding rich graphics functionality and interactivity to Java applications.

In our application, we display a series of graphs of available and target services, in a swing container; these graphs will be used in the construction of orchestrator.

### 1.1 Swing Components

#### *TabbedPane*

See a lot of information in a single window is inconvenient, so we chose to use a tabbed pane.

Each tabbed pane allows managing one type of information:

- ✓ Available Service Panel shows and manages all graph of available services;
- ✓ Target Service Panel shows the graph of target service;
- ✓ Cartesian Product Panel shows the graph of the Cartesian product of available services;
- ✓ Orchestrator Panel shows the simulation and the orchestrator in a table;
- ✓ Executable allows executing the orchestrator.

#### *Menu, Buttons and Labels*

For the controls we used simple components such as buttons and labels: labels display simple text messages and buttons are used to show information or to open dialog windows. In the menu bar we have the commands that control the whole system and not just a single graph or table, as a start.

#### *Dialog*

When you need to make choice, better way is to use a dialog. We have used dialogs to choose files, to save or load and to choose options.

#### *JGraph*

To show graph, we have used a library named JGraph. This library provides a class (JGraph) that inherits the class Component and allows drawing graphs.

#### *List and Table*

When we need to show a list or a table, we use JList or JTable to display lists or tables. These classes use a class Model that contains the data to show.

#### *JScrollPane*

When tables or graph component contain object largest window, we need panels with scroll bar: the JScrollPane container.

## Chapter 2: Panels

### 2.1 Available Panel and Target Panel

These panels are used to display graph associated to services. The buttons *AddNewService* open a file chooser to load files that contains services. The parser *ReadFile* reads the files and builds the objects *Service*. These objects now are passed to a class named *GraphFactory* that build a *JGraph* object and fills it with a vertex for each *State* and an edge for each *Action* of service.

You can also delete one service with the button *Remove*.

We can have many available services then available service panel have a list that contains the services loaded; when you click on one service in the list, the graph associated to the service is displayed on the screen. This panel also has the button *Remove All* that removes all services loaded.

The buttons *Save to Image* and *Save to DOT* allow saving the graph in Jpeg file or in a text file written in DOT format. The class *Exporter* has method to write images and text in File object.

### 2.2 Cartesian Product Panel

The Cartesian product of available service is a *Service* created by *CartesianProduct* class. It is an expensive operation so we display a message that says it.

Like for services, Cartesian product panel has the button *Save To Image* and *Save To DOT*. The panel also has the button *Change Representation* that allows changing the layout of the graph through the *JGraphLayout* library.

To choose the layout we use the dialog *ChoiceLayoutDialog*.

### 2.3 Orchestrator Panel and Execute Orchestrator Panel

When target service and at least one available service are loaded, you can click on *Start* in menu. This creates the simulation and the orchestrator.

In the orchestrator panel you can see simulation and orchestrator in the tables. These tables can be saved in an xml files with the button *Save To XML*. To create the tables, we use the class *XmlTableFactory* that transforms the objects *Orchestrator* and *Simulation* in Model for JT able and String in xml format.

To Execute orchestrator panel allow executing the orchestrator. In this panel are displayed the graph of target service, the initial states of target and available services, the Action of the current state and the service reachable from the initial state and the selected action.

The actions are taken from target service object; the services reachable with the action are taken using the objects *SimulatedBy* and *Orchestratorkey*.

Current state of the available services is contained in a *Cstate* object.

The maps *stateTable* and *actionTable* are used to color the vertex and the edge associated with current state and selected action.

## **2.4 The libraries JGraph and JGraphLayout**

JGraph is an open source graph visualization library written to be a fully Swing compatible component.

Graph cells can be drawn anywhere in a simple application, including on top of one another. Certain applications need to present their information in a generally ordered or specifically ordered structure. This might involve ensuring cells do not overlap and stay at least a certain distance from one another, or that cells appear in specific positions relative to other cells, usually the cells they are connected to by edges. This activity, called the layout application, can be used in a number of ways to assist users set out their graph.

The library JGraphLayout provides methods to apply layouts on a graph.

## **2.5 Configuration**

We need to display non-editable graphs (available and target service). Services have nodes that can be final or not final. The states are drawn as a circle and final states are drawn as double circle.

To draw circles and double circles we need override the class DefaultGraphCell, VertexView and VertexRenderer and, in particular, we need override the methods paint.

The graph target and available are not editable:

these properties are set in the class GraphFactory with the method JGraph.setEditable ()

The default color for the graphs is black; in panel execute orchestrator, the graph has the current node in blue and the selected edge in red.

---

2009, 15th May

## **Fourth Part: Test Cases**



## Chapter1: Test Cases

In this Chapter we report some test case with different number of services and changing states' number. These tests want show how more services are loaded and more the cost is high from a computational side. The important thing is the follow: we cannot know in how many time the orchestrator is calculated (with a big number of services) but the result is certainly correct. These test have been executed with a machine Intel Core2 Duo CPU @ 2.200GHz and RAM 2 GB.

We have decided to test application both in the case of the first implementation of community and in the last implementation of a "virtual community" with the same services. We experimented that in both cases the time taken is almost equal and we cannot express a fix time for each test due to the fact that each time record of simulation are checked in a different order. For this reason now we report a rough time obtained from the average of several tests executed in each case and that is the same in both version of community.

Let report some tests ordered by computational cost to build orchestrator :

- We added 3 available services with 10 states for each one and a target with 6 states so 6000 maximum number of simulation's record  $\Rightarrow$  to calculate orchestrator application spends about 4 seconds;

### Available Services

A1.txt	A2.txt	A3.txt
transition:	transition:	transition:
s1-a-s2;	q1-c-q2;	r1-c-r2;
s2-c-s1;	q1-b-q1;	r1-b-r3;
s1-a-s3;	q2-a-q3;	r1-a-r1;
s3-b-s4;	q2-b-q4;	r3-a-r1;
s4-a-s5;	q2-c-q5;	r3-c-r4;
s5-b-s4;	q1-c-q6;	r3-d-r5;
s4-a-s6;	q5-d-q6;	r4-b-r8;
s6-c-s7;	q6-a-q7;	r5-c-r6;
s7-c-s6;	q1-b-q7;	r1-b-r6;
s7-a-s8;	q7-a-q8;	r6-a-r7;
s8-b-s9;	q7-b-q9;	r5-a-r8;
s9-a-s10;	q9-c-q10;	r8-c-r9;
s8-b-s9;	initial:	r6-c-r9;
initial:	q1;	r9-d-r10;
s1;	final:	initial:
final:	q1;	r1;
s1;	q3;	final:
s2;	q4;	r1;
s5;	q8;	r2;
s6;	q10.	r7;
s10.		r10.

2009, 15th May

Target Service

T.txt

transition:

t1-a-t2;

t2-b-t2;

t2-c-t3;

t3-a-t4;

t4-b-t5;

t5-a-t6;

initial:

t1;

final:

t1.

- Proving to the application 4 available services and a target with 10 states, so 100000 maximum number of simulation's record  $\Rightarrow$  to calculate orchestrator application spends about 9 minutes;

Available ServicesTarget Service

We used the previous available services adding the following

A4.txt

transition:

p1-a-p2;

p2-c-p1;

p1-a-p3;

p3-b-p4;

p4-a-p5;

p5-b-p4;

p1-a-p6;

p6-c-p1;

p1-a-p7;

p7-b-p10;

p7-a-p8;

p8-b-p9;

initial:

p1;

final:

p1;

p5.

T2.txt

transition:

t1-a-t2;

t2-b-t2;

t2-c-t3;

t3-a-t4;

t4-b-t5;

t5-a-t6;

t6-d-t5;

t5-c-t7;

t4-d-t8;

t8-d-t10;

t8-c-t9;

initial:

t1;

final:

t1;

t6;

t7;

t9;

t10.

2009, 15th May

- Differently with 5 available services and target with 6 states, so 600000 maximum number of simulation's record  $\Rightarrow$  to calculate orchestrator application without virtual memory spends 18 hours differently application with virtual memory spends 18 hours and half .

Available ServicesTarget Service

We used the previous available services adding the following

A5.txt

transition:

u1-a-u2;

u2-c-u1;

u2-b-u9;

u2-d-u7;

u1-a-u3;

u3-b-u4;

u4-a-u5;

u5-b-u4;

u4-a-u6;

u7-c-u6;

u7-a-u8;

u8-b-u9;

u9-a-u10;

u8-b-u9;

u10-c-u5;

initial:

u2;

final:

u1;

u5;

u6.

T1.txt

transition:

t1-a-t2;

t2-b-t2;

t2-c-t3;

t3-a-t4;

t4-b-t5;

t5-a-t6;

initial:

t1;

final:

t1;

t6.

Let report some tests ordered by computational cost for Cartesian product algorithm:

- We added three available(A1.txt, A2.txt) services to application $\Rightarrow$  the calculus spends 32 msec;
- Providing to the system previous service and adding A3.txt the Cartesian product spends 4 minute.

These files are present in the root directory of application: Example's File folder.