University of Rome “La Sapienza”

FACULTY OF ENGINEERING
COURSE IN COMPUTER ENGINEERING

“Symphony 2.0”

Teacher: Giuseppe De Giacomo

Authors: Sciortino Giovanni rarefatto@gmail.com

Rome : 16 / 4 / 2009

Introduction

In this paper I describe the improvements introduces in the version 2.0 of Symphony respect to the version 1.0 of this program. To obtain more information about the architecture, the data structure used and the user manual of this program may be useful to read the documentation of Symphony 1.0.
In Symphony 2.0 I inserted a new methods to obtain the automatic composition of services. The user interface and the low-level structure are the same of the previous version except to the entry "Options" in the menu Edit that allow to select between the two different algorithm for obtain the composition.
This two algorithm are named:

- **Standard:** the original algorithm used in the version 1.0 that is described in documentation of Symphony 1.0
- **Plus:** an algorithm with the same goals of the previous but use a different approach that is described in this document.

In the following paragraph I describe the algorithm Plus.
The "Composition of services" algorithm that use the simulation relation may be divided in three main parts that are executed in sequence:

1. Creation of the asynchronous product of the available services
2. Compute the largest simulation between the target transition system and the asynchronous product of the available services generated in the first step.
3. Use the result of the second step and the description of the transition system to generate the function:

$$\mathrm{W}(\mathrm{target\ state}, \mathrm{states\ of\ the\ available\ services}, \mathrm{action}) = \{\mathrm{TS}_i \mid \forall \quad \mathrm{TS}_i \; can \quad execute \quad the\, action \quad a\}$$

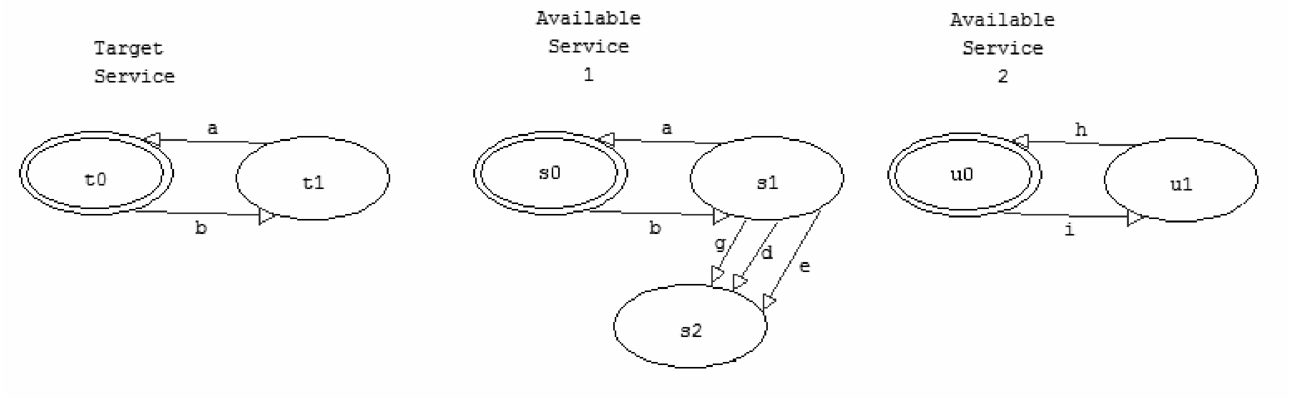
The description of the Plus orchestrator in the following paragraph is based on this partition of the Composition problem that will be called step 1, step 2 and step 3.

Plus orchestrator

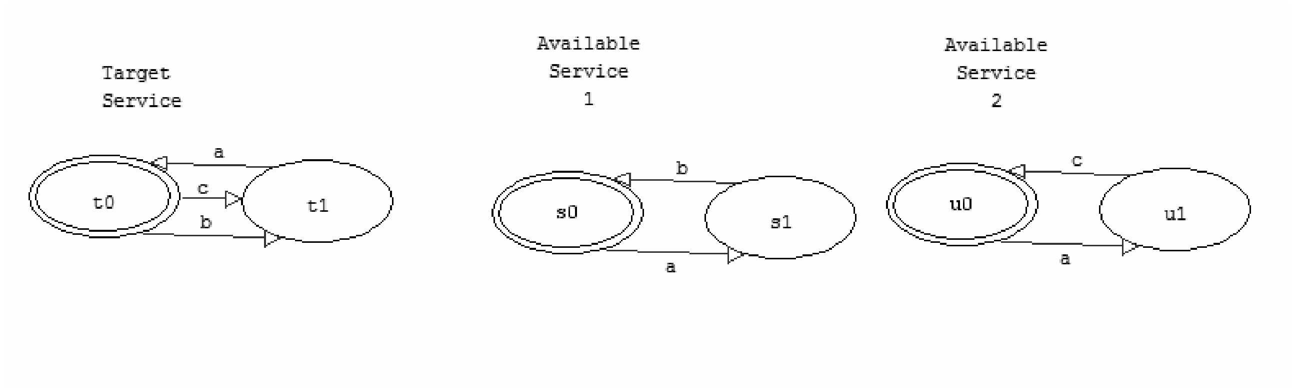
Introduction

This algorithm is a evolution of the previous algorithm used by Symphony 1.0, it executes the "Composition by simulation" using a smart approach. The previous algorithm, using several available services as input, requires several time to complete the execution because use a raw approach. Now I describe a list of example of input that slow down the execution of the previous algorithm:

1. The available services may be execute action that aren't used by the target system

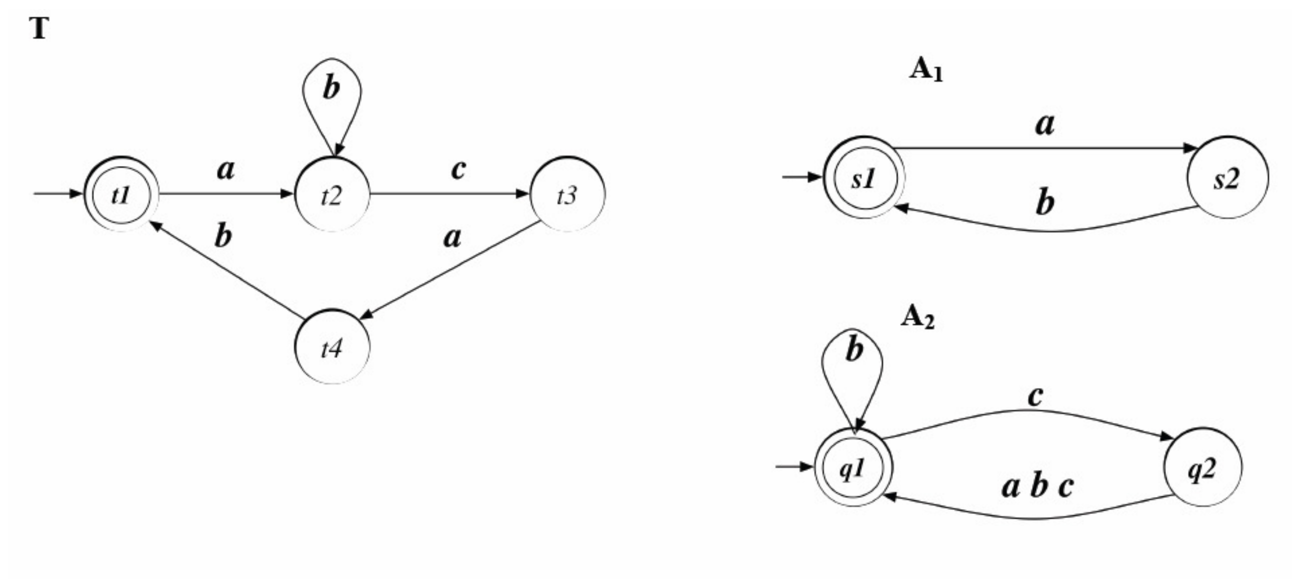


2. The available service may be use the same set of actions used by the TS but in a different order.



3. The pseudo code described in the page five of http://www.dis.uniroma1.it/~degiacom/didattica/software-services/lectures/Lecture05/4-composition-via-simulation-2up.pdf and used by the previous algorithm assert that every state of the target state is simulated by every state of the asynchronous product and then remove the non valid couple, the initial set of couple can be generated in other ways not inserting couples that we are sure that aren't contained in the simulation because for example any valid execution of the orchestrator generate them.

4. Another problem of the tradition "Composition by simulation" approach is that produce the largest simulation relation but some of these couple are unuseful for our goals. For example we can see the exercise(see the following image) proposed in the exam of the course "Service Integration" in the date 23/01/2009.



The biggest simulation relation that can be computed is:

```
(t1,s1q1)
(t2,s1q1)
(t2,s1q2)
(t2,s2q1)
(t3,s1q1)
(t3,s1q2)
(t3,s2q2)
(t4,s1q1)
(t4,s1q2)
(t4,s2q1)
```

But for each possible sequence of action executed from the target service some of these states of target service and available service will be never reached.
The couple of the biggest simulation relation that may be reached are:

```
(t1,s1q1)
(t2,s1q1)
(t2,s2q1)
(t3,s1q2)
(t3,s2q2)
(t4,s1q1)
(t4,s2q1)
```

The following three states never will be reached:

```
(t2,s1q2)
(t4,s1q2)
(t3,s1q1)
```

The state (t2,s1q2) will never reached because there are only two way to reach the state t2 of the target service:

- Executing a from t1 but in this case we reach the state (t1,s2q1) and not the state (t1,s1q2)
- Executing b from t2 but in this case we should remain in the state (t1,s1q2)

For this reason the state (t2,s1q2) is unreachable, may be demonstrated in the same way that also (t3,s1q1) and (t4,s1q2) are unreachable. Using the standard approach also these unreachable state will be used to calculate the function w (in the step 3) executing a computation that never will be used.

The smart approach to “Composition by simulation” that I have implemented in the orchestrator called “Orchestrator Plus” remove these redundancies.

Step 1: Create the asynchronous product

In this step there are the most relevant differences respect to the previous approach.
The algorithm compute a “pruned asynchronous product”, it doesn’t add every state that never can be reached executing any sequence of action with the target service. It also compute a set of couple of target state and available service state that are used as input in the step 2, the size of this set is minor or equal respect to the size of the initial set created with the standard approach because contain only the couple that may be reached executing any sequence of action on the target service.
The algorithm to execute to produce the “pruned asynchronous product” try all the possible sequence of actions that may be executed by the target service(detecting and removing infinite loops) and storing the effect of this action on available service’s states.

The algorithm use to store useful information the class SimilarState, that allow to store a couple of states:

- A state of the target service
- A state of the available service

This algorithm use two set to store object of the class SimilarState “setExpandedStates” and “setNotExpandedStates” and a set to store action “setExpandedAction”, initially all the sets are empty except to the set SetNotExpandedStates that contain a SimilarState’s object that contain the initial state of the target service and a ComposedState’s Object that represent the initial states of available services.
Then until the setNotExpandedStates is empty it will executed the following actions:

- Extract and remove a SimilarState from the set setNotExpdadedStates
- The SimilarState extracted is inserted in the set SetExpandedStates
- The SimilarState extracted contain the current state of target service and the current states of all the available service, so it does a iteration over all the action can be executed by the current target service’s state and check if each available service from the current state can execute the action. If it can, the algorithm compute create a new ComposedAction associated to this available service and the SimilarState that is reached executing this action updating the state of the service state and the state of one available service. Then the ComposedAction is inserted in the setExpandedAction and the SimilarState computed is inserted in the set setNotExpandedState only if the Similartate isn’t contained already in one of the two set used to store SimilarState(it does this check to avoid infinite loops)

Finally all the states and actions contained in the sets setExpandedStates and setExpandedActions are inserted in the Transition System that describe the pruned asynchronous product. The output of this step is composed by the pruned asynchronous product and from the set setExpandedState. The set setExpandedState is used also as initial set of the step 2 because it contains all the state that can be reached from the initial state but we don’t know without executing the simulation’s algorithm if the execution of a action’s sequence starting from each of these state will bring the system to a valid state, but in any case the computation’s time used is equal or minor respect to the traditional approach.

```
public static ResultPrunedAsyncProduct asynchronousProductPruned(

TransitionSystem<SimpleState, SimpleTransition> target,

TransitionSystem<SimpleState, SimpleTransition>[] availableServices,
                                        ServiceFactory serviceFactory
                                                  ) throws Exception {
      //set the initialState of all the services
      SimpleState initialStates[]=new
            SimpleState[availableServices.length];
      for(int i=0;i<availableServices.length;i++){
            initialStates[i]=availableServices[i].getInitial();
      }
      ComposedState initialState=
            serviceFactory.createComposedState(initialStates);
      SimpleState initialStateTarget=target.getInitial();
      //create setStateNotExpanded that contain all the state
      //that will be examined
      HashSet<SimilarState> setStateNotExpanded=
            new HashSet<SimilarState>();
      //create setStateExpanded that contain all the state was already
      //checked
      HashSet<SimilarState> setStateExpanded=new HashSet<SimilarState>();
      HashSet<ComposedTransition> setStateExpandedAction=
            new HashSet<ComposedTransition>();
      setStateNotExpanded.add(
            new SimilarState(initialStateTarget,initialState)
      );
      //while there are state not expanded
      while(!setStateNotExpanded.isEmpty()){
            SimilarState currentSimilar=
                  setStateNotExpanded.iterator().next();

            setStateNotExpanded.remove(currentSimilar);
            //check if the target state is final and the ComposedState
            //isn't final, in this case remove it
            if(currentSimilar.getTargetState().isFinal() &&
                  !currentSimilar.getAvalableServiceState().isFinal())
                  continue;
            //check if the current ComposedState can execute all the
            //action that can be executed by the target
            boolean allActionSimulated=true;
            if(setStateExpanded.contains(currentSimilar))
                  continue;
            setStateExpanded.add(currentSimilar);

            HashSet<SimilarState> candidatesForSimilar=
                  new HashSet<SimilarState>();
            HashSet<ComposedTransition> candidatesForSimilarActions=
                  new HashSet<ComposedTransition>();

            SimpleState currentTargetState=
                  currentSimilar.getTargetState();
            ComposedState currentComposedState=
                  currentSimilar.getAvalableServiceState();
            Iterator<SimpleTransition> itTargetStateAction=
                  target.getActionsOf(currentTargetState).iterator();
            while(itTargetStateAction.hasNext() &&
                  allActionSimulated==true){
                  boolean thisActionSimulated=false;
                  SimpleTransition
                  currentTargetAction=itTargetStateAction.next();
                  for(int i=0;i<currentComposedState.getStates().length &&
                              allActionSimulated==true;i++){
                              Iterator<SimpleTransition>
                                   itActionsOfThisComposedState=
                                   availableServices[i].getActionsOf(
              (SimpleState)currentComposedState.getServicesState(i)
                              ).iterator();
                              while(itActionsOfThisComposedState.hasNext()){
                                   SimpleTransition
                   currentAction=itActionsOfThisComposedState.next();

          if(currentAction.getName().equals(currentTargetAction.getName())){
               //add this couple to the pruned asyncronous product
               State[] newStates=
                    currentComposedState.getStates().clone();

               newStates[i]=currentAction.getStateTo();
               SimpleState nextTargetState=
                    (SimpleState)currentTargetAction.getStateTo();
               ComposedState nextComposedState=
                    serviceFactory.createComposedState(newStates);
               SimilarState newCandidate=
                    new SimilarState(nextTargetState,nextComposedState);
               candidatesForSimilar.add(newCandidate);
               ComposedTransition candidateAction=
                    serviceFactory.createComposedAction(
                         currentTargetAction.getName(),
                         availableServices[i],
                         currentComposedState,
                         nextComposedState

                      );
               candidatesForSimilarActions.add(candidateAction);
               thisActionSimulated=true;
          }
     }
     }
     if(!thisActionSimulated){
          allActionSimulated=false;
     }
     }
     if(allActionSimulated){
          Iterator<SimilarState> itCandidateSimilar=candidatesForSimilar.iterator();
          while(itCandidateSimilar.hasNext()){
               SimilarState currentSimilarState=itCandidateSimilar.next();
               if(!setStateExpanded.contains(currentSimilarState)){
                    setStateNotExpanded.add(currentSimilarState);
               }
          }
          Iterator<ComposedTransition> itCandidateSimilarAction=
               candidatesForSimilarActions.iterator();
          while(itCandidateSimilarAction.hasNext()){
               ComposedTransition currentCandidateSimilarAction=
                    itCandidateSimilarAction.next();
          setStateExpandedAction.add(currentCandidateSimilarAction);
          }
     }
     }
     TransitionSystem<ComposedState,ComposedTransition> result=
          serviceFactory.createComposedTransitionSystem();
     Iterator<SimilarState> itSimilarState=setStateExpanded.iterator();
          while(itSimilarState.hasNext()){
               try{
          result.addState(itSimilarState.next().getAvalableServiceState());
               }catch(Exception e){
			}
		}
		Iterator<ComposedTransition> itAction=setStateExpandedAction.iterator();
			while(itAction.hasNext()){
				ComposedTransition currentAction=itAction.next();
				result.addAction(currentAction.getStateFrom(),
					currentAction.getStateTo(), currentAction);
			}

			return new ResultPrunedAsyncProduct(result,setStateExpanded);

		}
```

Step 2: Compute the simulation

The step two (to calculate the largest simulation) is computed with an algorithm that implements the pseudo code described in the page five of http://www.dis.uniroma1.it/~degiacom/didattica/software-services/lectures/Lecture05/4-composition-via-simulation-2up.pdf
The algorithm works on couple of states, one SimpleState (a state of the target system) and a ComposedState (the states of the available services), to represent this couple I used the class SimilarState that contain as attributes a SimpleState and a ComposedState and define standard java's methods as equals and hashCode. I implement the methods equals and hashCode for the class Similar State because the algorithm works on a large set of couple that is frequently accessed to check if a couple exists in the simulation relation so I used a HashSet
(http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashSet.html) to store this set and access to its element and check if exists a couple in this set in constant time.
Finally the algorithm generate the function w(step 3) described in the introduction, it use as input the largest simulation relation obtained in the previous step, the description of the target service and the asynchronous product obtained in the step 1 pruned deleting every state that doesn't simulate any state of the target service.
The result of the step 2 is a set of SimilarState objects, for each SimilarState SS_i extracted we can identify two part a state of the target service $SStarget_i$ and a state of the asynchronous product $SSasyncProd_i$.
The algorithm execute a iteration over all the couple of the set of SimilarState for each SS_i it search which action may be executed from the state $SStarget_i$ and for each of them compute which transition system can be execute it extracting them from the outing actions of the state $SSasyncProd_i$.
May be occur that a available service is non deterministic, in this case the algorithm check, before to assert that this TS can execute the action, that the execution of any of these actions brings the state of the available service in valid states to follow the execution.
In this step the algorithm also create the composition Transition System, it contain all the state of the pruned asynchronous product that are contained in one ore more couple of the simulation's result.

```
		public static SimulationPlusImpl generate(
		TransitionSystem<SimpleState, SimpleTransition> target,
		TransitionSystem<SimpleState, SimpleTransition>[] availableServices,
		ServiceFactory serviceFactory)
			throws Exception{


			//get the async-product result as a transition system
			//that is composed by the pruned async product and the couples
			//candidate for the simulation
```

```
        ResultPrunedAsyncProduct resultAsyncPrunedProd=
        asynchronousProductPruned(target,availableServices,serviceFactory);
        HashSet<SimilarState> r=
        resultAsyncPrunedProd.getCandidatesForSimulation();
        TransitionSystem<ComposedState, ComposedTransition> asyncPrunedProd=
            resultAsyncPrunedProd.getPrunedAsyncProd();
        //while r was modified in the last iteration
        boolean rModified=true;
        while(!r.isEmpty() && rModified==true){
            //this set contain couples that will be removed at the end of
            //the iteration
            HashSet<SimilarState> coupleToRemove=
                new HashSet<SimilarState>();
            rModified=false;
            //iterator over the elemen of t
            Iterator<SimilarState> itCurrentSimilarCouple=r.iterator();
            while(itCurrentSimilarCouple.hasNext()){
                //currentSimilarCouple is the current couple of state
                //that will be checked
                SimilarState currentSimilarCouple =
                    itCurrentSimilarCouple.next();
                Iterator<SimpleTransition>
                itTargetActions=target.getActionsOf(
                    currentSimilarCouple.getTargetState()
                    ).iterator();
                boolean existSimulation=true;
                //check if the available state can be execute every
                //action that can be
                //executed by the target service
                while(itTargetActions.hasNext() &&
                    existSimulation==true){
                    existSimulation=false;
                    SimpleTransition currentTargetAction=
                    itTargetActions.next();

                    Iterator<ComposedTransition>
                        itActionsOfComposedState=
                        asyncPrunedProd.getActionsOfStateWithName(

                    currentSimilarCouple.getAvalableServiceState(),

                    currentTargetAction.getName()

                    ).iterator();
                    while(itActionsOfComposedState.hasNext()){

                        ComposedTransition
                        currentActionComposedState=
                        itActionsOfComposedState.next();
                SimilarState temp=new SimilarState(
                (SimpleState)currentTargetAction.getStateTo(),
                currentActionComposedState.getStateTo()
                );
                if(r.contains(temp))
                    existSimulation=true;
                }
            }
//if the couple isn't in the simulation, add the couple in the set of couples
//that will be removed
            if(!existSimulation){
                coupleToRemove.add(currentSimilarCouple);
            }
        }
        if(coupleToRemove.size()>0){
            rModified=true;
        }
    }
    //create the composition TS
    //contains only the states(of the available service)that are in the set r
    TransitionSystem<ComposedState, ComposedTransition>
        resultPrunedWithSimulation=
            serviceFactory.createComposedTransitionSystem();
        Iterator<SimilarState> itR=r.iterator();
        while(itR.hasNext()){
            ComposedState currentState=
                itR.next().getAvalableServiceState();
            try{
                resultPrunedWithSimulation.addState(currentState);
            }catch (Exception e) {
            }
        }
        //for each state, add a transition only if the target service does
        //it
        Iterator<SimilarState> itSimulationData=r.iterator();
        while(itSimulationData.hasNext()){
            SimilarState simulationCouple=itSimulationData.next();
            SimpleState currentSimpleState=
                simulationCouple.getTargetState();
            Iterator<SimpleTransition> itSimpleAction=
                target.getActionsOf(currentSimpleState).iterator();
            while(itSimpleAction.hasNext()){
                SimpleTransition currentSimpleAction=
                    itSimpleAction.next();
                ComposedState composedState=
                    simulationCouple.getAvalableServiceState();
                Iterator<ComposedTransition> itComposedAction =
                resultAsyncPrunedProd.getPrunedAsyncProd().getActionsOfS
                tateWithName(composedState,currentSimpleAction.getName()
                ).iterator();
                while(itComposedAction.hasNext()){
                    ComposedTransition currentComposedAction=
                        itComposedAction.next();
                    SimilarState candidatesForSimulation=
                        New SimilarState(
                    (SimpleState)currentSimpleAction.getStateTo(),
                     currentComposedAction.getStateTo());
                    if(r.contains(candidatesForSimulation)){
                        Set<ComposedTransition> composedActionOther=
                    getActionsOfStateWithNameAndTSandStateFrom(
                    resultAsyncPrunedProd.getPrunedAsyncProd(),
                    currentComposedAction
                    );
                    if(composedActionOther.size()==1)
            resultPrunedWithSimulation.addAction(
                currentComposedAction.getStateFrom(),
                currentComposedAction.getStateTo(),
                currentComposedAction
            );
            else {
                boolean toAdd=true;
                Iterator<ComposedTransition> itOtherComposedAction=
                    composedActionOther.iterator();
                while(itOtherComposedAction.hasNext()){
                    ComposedTransition otherComposedAction=
                        itOtherComposedAction.next();
                SimilarState otherCandidateForSimulation=
        new SimilarState((SimpleState)currentSimpleAction.getStateTo(),
            otherComposedAction.getStateTo());
            if(!r.contains(otherCandidateForSimulation)){
                    toAdd=false;
            }
    }
    if(toAdd){
            resultPrunedWithSimulation.addAction(
                    currentComposedAction.getStateFrom(),
                    currentComposedAction.getStateTo(),
                    currentComposedAction
            );
    }
}
}
}
}

}

SimulationPlusImpl result=new SimulationPlusImpl();
            result.target=target;
            result.composition=resultPrunedWithSimulation;
            result.asyncProd=resultAsyncPrunedProd.getPrunedAsyncProd();
            result.relation=r;
            return  result;
}
```

Step 3: Compute the transition function

Using the composition transition system generated in the step 2, create the transition function is easy because it contain all the information that we need.
The composition Transition System has states that are couple of a target state and the available service state and edge that contain the name of the action and which TS can execute it.

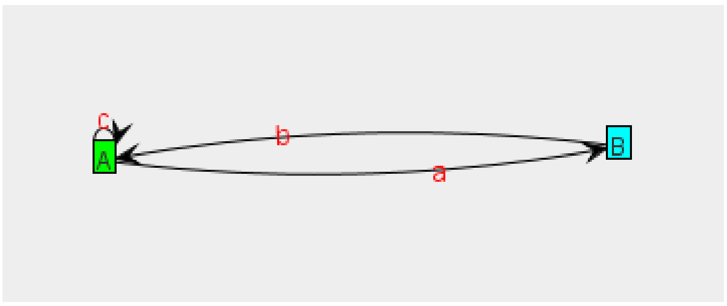
```
private Vector<TransitionResultEntry> createTransitions() throws
        Exception{
                Vector<TransitionResultEntry> result=
                        new Vector<TransitionResultEntry>();
                TransitionSystem<ComposedState, ComposedTransition> compositionTS=
                        getCompositionTs();
                Set<SimilarState> setSimilarState=sym.getAllSimilarState1();
                Iterator<SimilarState> itSimulationCouple=
                        setSimilarState.iterator();
                while(itSimulationCouple.hasNext()){
                        SimilarState currentSimulationCouple=
                                itSimulationCouple.next();
                        Iterator<SimpleTransition> itSimpleAction=
        target.getActionsOf(
                currentSimulationCouple.getTargetState()
        ).iterator();
        while(itSimpleAction.hasNext()){
                SimpleTransition simpleAction=itSimpleAction.next();
                Iterator<ComposedTransition> itComposedAction=
                compositionTS.getActionsOfStateWithName(
                        currentSimulationCouple.getAvalableServiceState(),
                        simpleAction.getName()
                ).iterator();
                TransitionResultEntry newTransition=
                        new TransitionResultEntry(
                                currentSimulationCouple.getTargetState(),
                                currentSimulationCouple.getAvalableServiceState(),
                                simpleAction
```

```
			);
		while(itComposedAction.hasNext()){
			ComposedTransition currentComposedAction=
				itComposedAction.next();

		newTransition.addTransitionSystemInTransition(
			(TransitionSystem<SimpleState, SimpleTransition>)
			currentComposedAction.getService()
		);
	}
	result.add(newTransition);
}
}
Collections.sort(result);
return result;
}

public class TransitionResultEntry implements Comparable<TransitionResultEntry>{
	private SimpleState targetState;
	private ComposedState composedState;
	private String action;
	private Vector<TransitionSystem<SimpleState,SimpleTransition>> ts;
	public TransitionResultEntry(SimpleState targetState,ComposedState
		composedState,Transition action){
		this.targetState=targetState;
		this.composedState=composedState;
		this.action=action.getName();
		this.ts=
			new Vector<TransitionSystem<SimpleState,SimpleTransition>>();
	}
	public TransitionResultEntry(SimpleState targetState,
		ComposedState composedState,Transition action,
		TransitionSystem<SimpleState, SimpleTransition> ts){

		this.targetState=targetState;
		this.composedState=composedState;
		this.action=action.getName();
		this.ts=
			new Vector<TransitionSystem<SimpleState,SimpleTransition>>();
		this.ts.add(ts);
	}
	public TransitionResultEntry(SimpleState targetState,
		ComposedState composedState,
		Transition action,
		Set<TransitionSystem<SimpleState, SimpleTransition>> ts){
		this.targetState=targetState;
		this.composedState=composedState;
		this.action=action.getName();
		this.ts=
			new Vector<TransitionSystem<SimpleState,SimpleTransition>>();
		this.ts.addAll(ts);
	}
	public void addTransitionSystemInTransition(TransitionSystem<SimpleState,
SimpleTransition> ts){
		//check if already exist a ts with the same name
		Iterator<TransitionSystem<SimpleState,SimpleTransition>>
			itTS=this.ts.iterator();
		while(itTS.hasNext())
			if(itTS.next().getName().equals(ts.getName()))
				return;
		this.ts.add(ts);
	}
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((action == null) ? 0 :
			action.hashCode());
		result = prime * result
				+ ((composedState == null) ? 0 :
					composedState.hashCode());
		result = prime * result
				+ ((targetState == null) ? 0 : targetState.hashCode());
		return result;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		TransitionResultEntry other = (TransitionResultEntry) obj;
		if (action == null) {
			if (other.action != null)
				return false;
		} else if (!action.equals(other.action))
			return false;
		if (composedState == null) {
			if (other.composedState != null)
				return false;
		} else if (!composedState.equals(other.composedState))
			return false;
		if (targetState == null) {
			if (other.targetState != null)
				return false;
		} else if (!targetState.equals(other.targetState))
			return false;
		return true;
	}
	public String[] toStringArray(){
		String[] result=new String[]{
			targetState.getName(),
			"",
			action,
			""
		};
		for(int i=0;i<composedState.getStates().length;i++){
			try {
				if(i>0)
					result[1]=result[1]+",";

	result[1]=result[1]+composedState.getServicesState(i).getName();
			} catch (Exception e) {
				e.printStackTrace();
			}
		}
		for(int i=0;i<ts.size();i++){
			if(i>0)
				result[3]=result[3]+",";
			result[3]=result[3]+ts.elementAt(i).getName();
		}
		return result;
	}
	/* (non-Javadoc)
	 * @see java.lang.Comparable#compareTo(java.lang.Object)
	 */
```

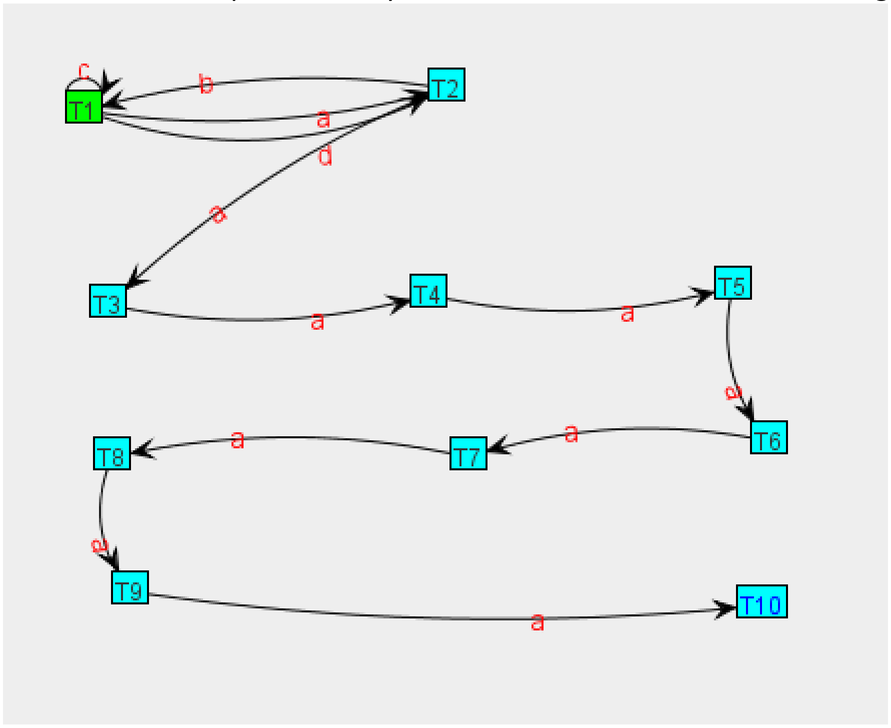
```
        @Override
        public int compareTo(TransitionResultEntry t) {
              int
    targetName=targetState.getName().compareTo(t.targetState.getName());
              if(targetName!=0)
                    return targetName;
              int
    composedStateName=composedState.getName().compareTo(t.composedState.getName());
              if(composedStateName!=0)
                    return composedStateName;
              int actionName=action.compareTo(t.action);
              if(actionName!=0)
                    return actionName;

              return 0;
        }

    }
```

Performance

I tested the performance of these three approach using the example “exampleBig” that is also provided with the project. The target service used is:



And all the available services are represented by the same TS showed in the next image:



Any available service of this example has ten states, so using n available services the asynchronous product of the available service have 10^n states.
In the next table I express the average execution time(expressed in millisecond) to execute the three algorithms described in this document compared by the number of asynchronous product's states computed in the traditional "Composition by simulation approach".

# states in the asynchronous product	**Standard**	**Plus**
10^3	172	1
10^4	842	10
10^5	29032	19
10^6	More than 5 minutes	25

The previous test is only a toy example that allow to fully exploit the features of the Orchestrator Plus, may be useful in future works to test this two algorithm using real big instances or calculate the computation complexity of these algorithm.