Model Driven Service Composition

Content

- Motivation
- Cornerstones of our approach
- Functional requirements for service composition
- Information model for service composition
- Process of service composition development
- Conclusions
- Question and remarks

Motivation

- Platform neutral nature of web services creates the opportunity to develop business processes by using and combining existing web services

- Service composition is too complex and too dynamic to handle manually (e.g. a vast service space to search, a variety of services to compare and match, and different ways to construct composed services)

- However, current composite web service development and management solutions are very much a **manual** activity, which require specialized knowledge and take up much time and effort.

Cornerstones of our approach

- **We use a model driven approach to facilitate the development and management of dynamic service compositions**
  - Functional requirements
  - Information meta model
  - Architecture
  - Algorithm
- **To govern and steer the process of service composition development we utilise rules**
  - Classification
  - Specification
  - Application

Functional requirements

- **Service composition development**

  The application developer interacts with the service composition system to generate a business process by composing services. The use case starts when the developer sends a request. The system at the end produces an executable service composition.

- **Service composition management**

  The application developer interacts with the service composition system to execute and manage compositions. This use case begins when the developer indicates that he wants to execute a service composition. In response the system gathers the required information and subsequently executes the composition. During run-time the developer may interact with the service composition system to make modifications.

Service composition development

- **Definition phase**

  The system starts by defining a composite service in an abstract manner, e.g. with regard to offered functionality and constraints.

- **Scheduling phase**

  Next, the system determines how and when services should run and prepares them for execution.

- **Construction phase**

  Then, the system proceeds to construct an unambiguous composition of concrete services out of a set of desirable or potentially available/matching constituent services.

- **Execution phase**

  Lastly, the system prepares the constructed composed services for execution.

Information model

- Is an abstract meta-model that represents the building blocks of all possible service compositions.

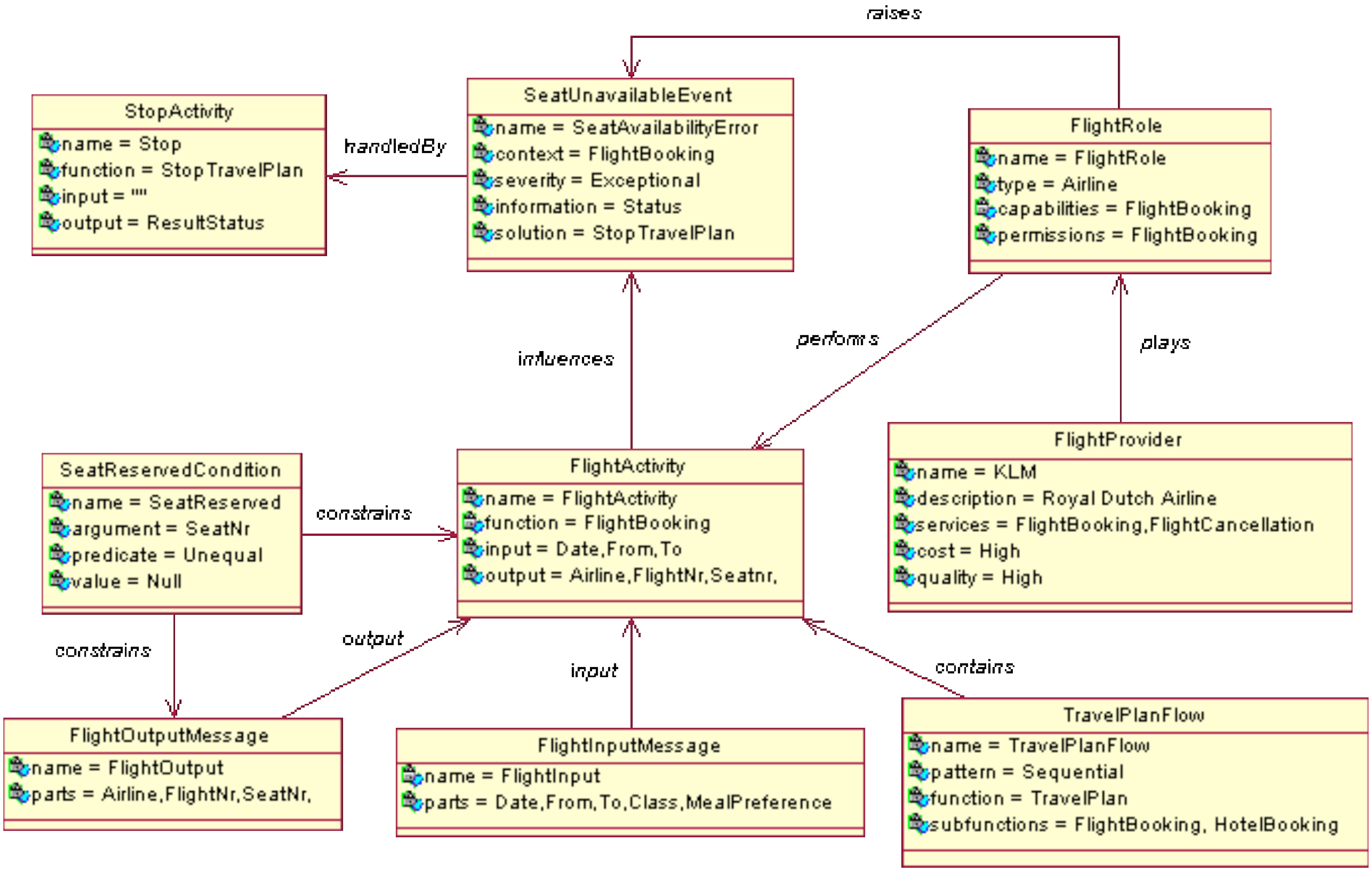
- Models the components required for a given composition as well as their inter-relationships. Relationships in the IM indicate how a composition is constructed.

- All the required information is represented as classes containing special purpose attributes, referred to as *composition classes.*

- Specific instances of the model are generated by populating its classes. Class instances are referred to as *composition elements.*

- Is expressed in UML to support the development of technology independent service composition definitions.

Sample IM instance



StopActivity
- name = Stop
- function = StopTravelPlan
- input = ""
- output = ResultStatus

SeatUnavailableEvent
- name = SeatAvailabilityError
- context = FlightBooking
- severity = Exceptional
- information = Status
- solution = StopTravelPlan

FlightRole
- name = FlightRole
- type = Airline
- capabilities = FlightBooking
- permissions = FlightBooking

SeatReservedCondition
- name = SeatReserved
- argument = SeatNr
- predicate = Unequal
- value = Null

FlightActivity
- name = FlightActivity
- function = FlightBooking
- input = Date,From,To
- output = Airline,FlightNr,Seatnr,

FlightProvider
- name = KLM
- description = Royal Dutch Airline
- services = FlightBooking,FlightCancellation
- cost = High
- quality = High

FlightOutputMessage
- name = FlightOutput
- parts = Airline,FlightNr,SeatNr,

FlightInputMessage
- name = FlightInput
- parts = Date,From,To,Class,MealPreference

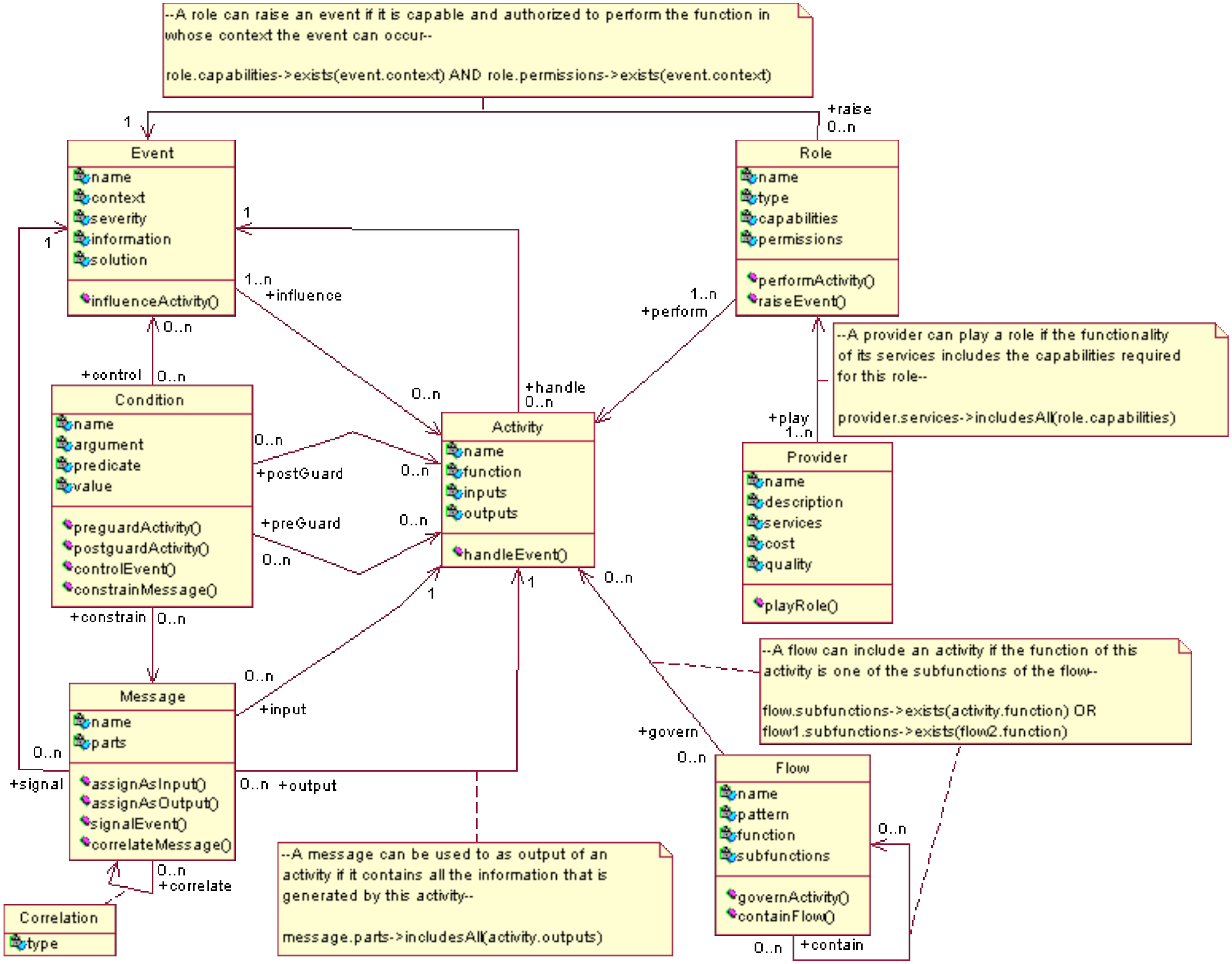
TravelPlanFlow
- name = TravelPlanFlow
- pattern = Sequential
- function = TravelPlan
- subfunctions = FlightBooking, HotelBooking

Service Composition Development Process

- **A concrete service composition needs to link elements such as "service provider" to “role", “role" to "activity", "activity" to "flow", and so on, to construct an instance of the service composition IM.**

- **These associations are constrained by means of rules, referred to as *composition rules***
  - Are often deeply embedded in application code, whereas
  - Extraction and explicitness increases ease of management (defined and versified) as well as execution consistency.

- **Composition rules are expressed in the Object Constraint Language (OCL). We apply these rules to constrain composition element attributes values and associations. In other words, we use them to drive the service composition development process.**



--A role can raise an event if it is capable and authorized to perform the function in whose context the event can occur--

role.capabilities->exists(event.context) AND role.permissions->exists(event.context)

--A provider can play a role if the functionality of its services includes the capabilities required for this role--

provider.services->includesAll(role.capabilities)

--A flow can include an activity if the function of this activity is one of the subfunctions of the flow--

flow.subfunctions->exists(activity.function) OR flow1.subfunctions->exists(flow2.function)

--A message can be used to as output of an activity if it contains all the information that is generated by this activity--

message.parts->includesAll(activity.outputs)

Rule classification

- **Structural rules**

  Guide the process of structuring, scheduling and prioritizing activities.

- **Behavioral rules**

  Specify conditions for the composition behavior, e.g to guard activities.

- **Data rules**

  Control the use of data by activities, data dependencies, and etceteras.

- **Resource rules**

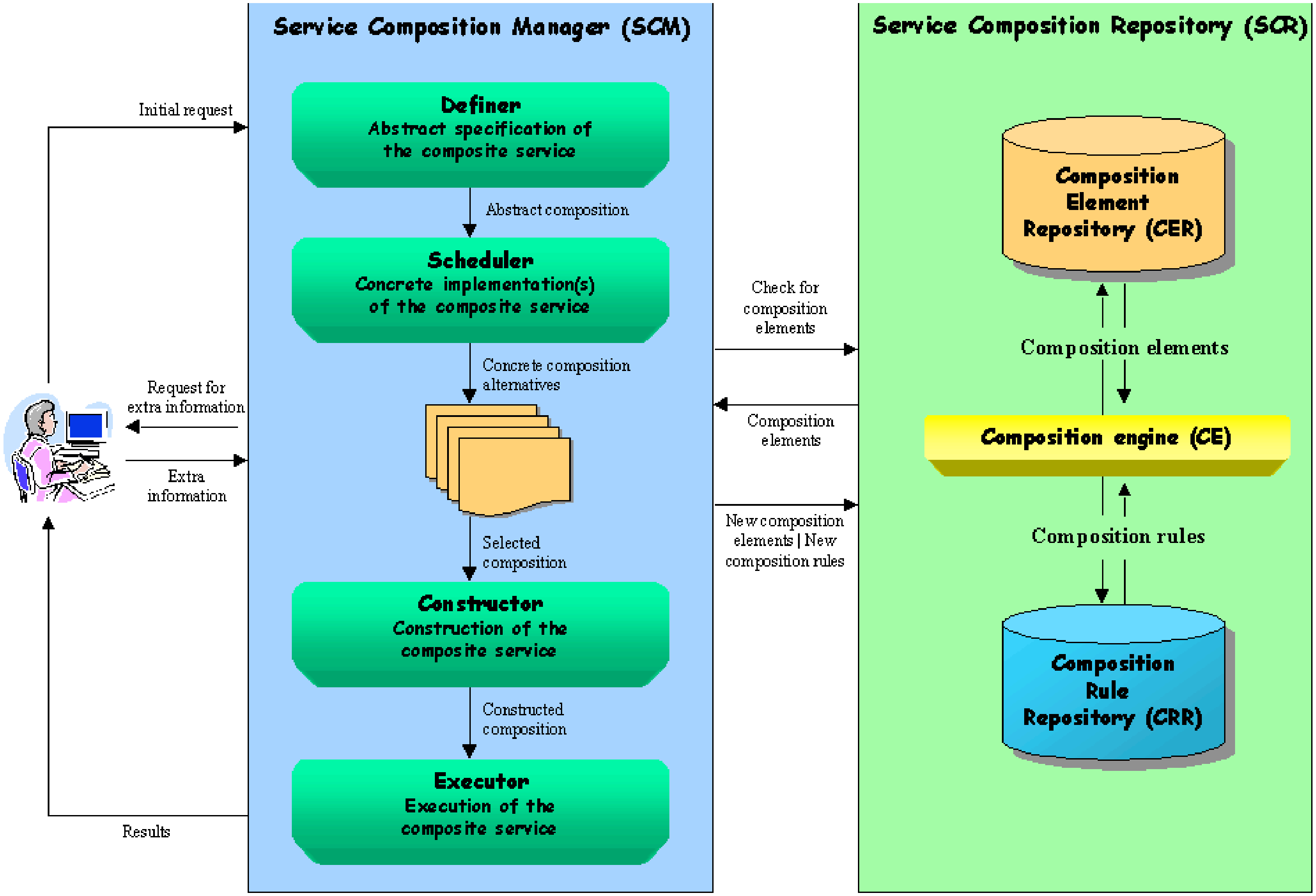
  Guide the use of resources, e.g. in terms of selecting providers.

- **Exception rules**

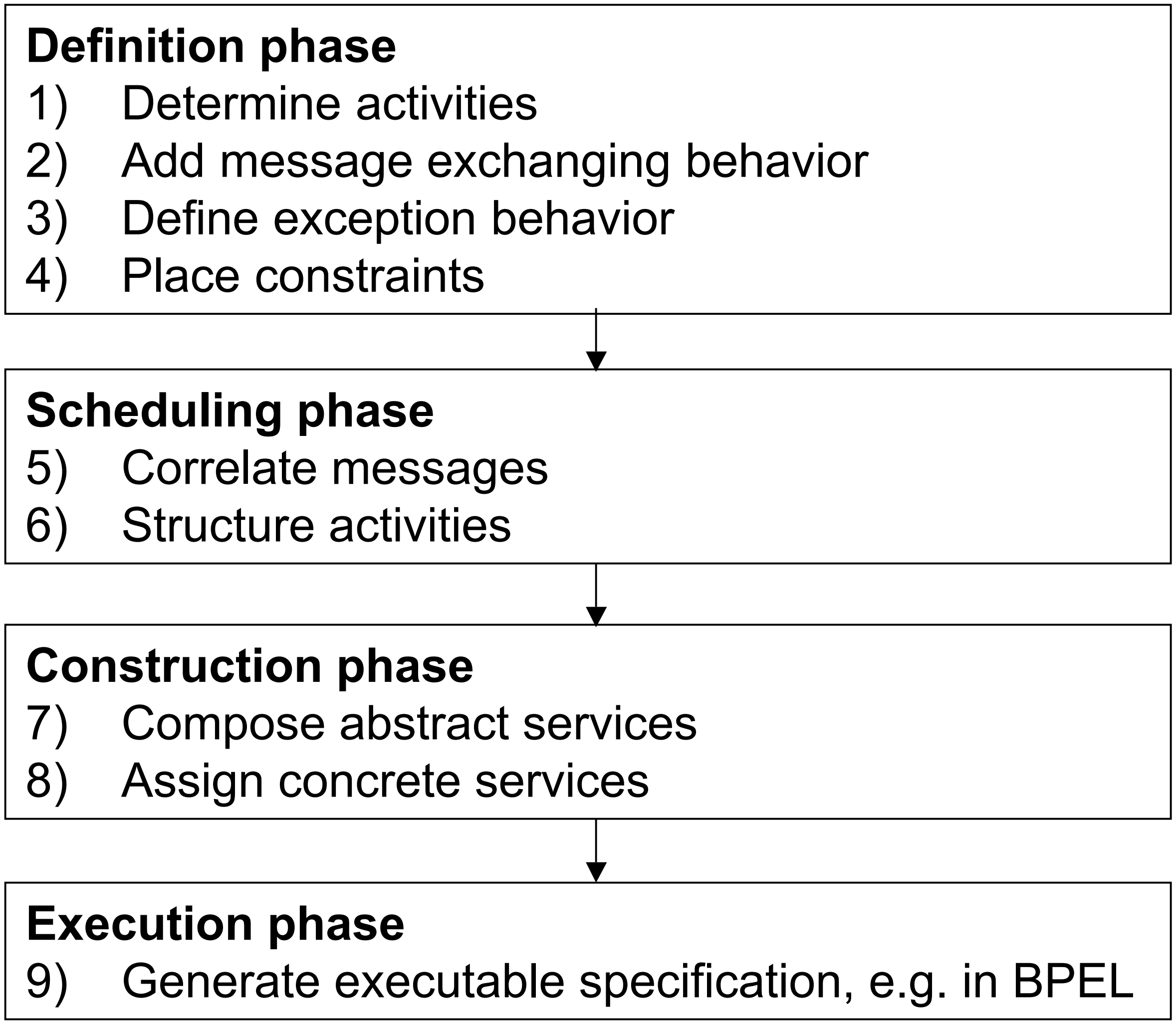
  Govern the exceptional behavior, e.g. fault handling.

UNIVERSITEIT VAN TILBURG

Service Composition Development System


Service Composition Manager (SCM)
Service Composition Repository (SCR)
Initial request
Definer
Abstract specification of the composite service
Abstract composition
Scheduler
Concrete implementation(s) of the composite service
Concrete composition alternatives
Request for extra information
Extra information
Selected composition
Constructor
Construction of the composite service
Constructed composition
Executor
Execution of the composite service
Results
Check for composition elements
Composition elements
New composition elements | New composition rules
Composition Element Repository (CER)
Composition elements
Composition engine (CE)
Composition rules
Composition Rule Repository (CRR)

System Algorithm



Definition phase

1) Determine activities
2) Add message exchanging behavior
3) Define exception behavior
4) Place constraints

Scheduling phase

5) Correlate messages
6) Structure activities

Construction phase

7) Compose abstract services
8) Assign concrete services

Execution phase

9) Generate executable specification, e.g. in BPEL

Examples

- **Add message exchanging behavior**

```
for each Activity
        do while (no Input for Activity)
              apply assignAsInput in Message to
              every Message/Activity combination
```

- **Definition of assignAsInput:**

  message.parts->includesAll(activity.inputs)

Examples, continued

- **Suppose we have the following elements:**
  - FlightActivity: inputs="Date,From,To"
  - Message1: parts="CheckinDate,Duration,HotelName"
  - Message2: parts="Date,ReturnDate,From,To,Class,MealPreference"

- **Application of assignAsInput then results in assigning Message2 to FlightActivity as this activity's input:**
  - First **Message1** is tried, but the resulting activity/message combination does not meet the requirements in the rule (not surprisingly, since this message contains hotel reservation info)
  - Subsequently a combination with **Message2** is tested; this combination is successful, because this message provides a superset of data required for **FlightActivity**.

Examples, continued

- **Compose abstract services**

```
for each Activity
        do while (no Role for Activity)
              apply performActivity in Role to every
              Role/Activity combination
```

- **Definition of performActivity:**

  role.capabilities->exists(activity.function)

  AND role.permissions->exists(activity.function)

Example, continued

- **Suppose we have the following elements:**
  - FlightActivity: function=“FlightBooking”
  - Role1: capabilities=“FlightBooking”, permissions=“FlightBooking”
  - Role2: capabilities=“CarRental”, permissions=“CarRental”

- **Application of performActivity then results in assigning Role1 to FlightActivity as this activity’s abstract service:**
  - When **Role1** is tried, the resulting activity/role combination meets the requirements in the rule and thus is suitable to be assigned to **FlightActivity** as its abstract service

- **Note: if there would not have been a suitable role, then the system would have consulted the user to provide the service requirements for FlightActivity.**

Conclusions

- **Current standards (e.g. BPEL, BPML) are not suitable for flexible and dynamic service composition.**

- **Our approach does cater this by:**

  - Using a model driven approach to facilitate the development and management of service compositions, allowing flexible and rapid development and delivery of service compositions based on proven and tested models, as such supporting the service composition life-cycle.
  - Applying rules to drive the service composition development process, as such paving the way towards developing dynamic service compositions.

Future research

- **Just to mention a few:**
  - Investigate and formally verify mapping and conformance between compositions
  - Design of the rule mechanism to manage and apply the composition rules in accordance with the defined algorithm
  - Development of a change system to manage the evolution of composition elements and rules, and service composition specifications

Questions/remarks

?