An agent based architecture for constructing Interactive Simulation Systems

Zhao, Z.

Citation for published version (APA):
Chapter 3

Agent based activity orchestration

In this chapter, we first give a functional description of the architecture and then discuss how the agents control the run-time system behaviour. We focus on a number of innovative designs in ISS-Conductor: capability and story modelling mechanisms, and scenario execution paradigms*.

3.1 An ISS as a multiple Module Agents system

First, we give a short review on the basic concepts introduced in the preceding chapter. An ISS-Conductor component has an Actor and a Conductor, which respectively encapsulate the functionalities of a simulation or visualisation system and control the run-time activities. Using ISS-Conductor components, an ISS is realised as a collection of component instances with different roles and a story of the interaction constraints between them. A story is designed based on the capabilities of the employed components. The Module Agents in the Conductors use the capability and story to control the run-time system behaviour.

As we have discussed in chapter two, an MA achieves its deliberative control on run-time activities using a reasoning kernel, as shown in Fig. 2.4. The knowledge base of the reasoning kernel not only contains the descriptions of story and capability, but also contains the control intelligence for reasoning on activities and for interacting with the other MAs. An MA can thus be described as a machinery which contains four functional components: a controller, a world model, a story and a capability, as shown in Fig. 3.1. The capability serves as a sort of expert system for answering “what can I actually do?”. The story indicates “what am I expected to do?”. The world model shows “what is going on in the environment?”. The controller realises general strategies for managing the world model and capability, for interpreting the contents of a story and for controlling the Actor.

The chapter is organised as follows. In 3.2 and 3.3, we describe the basic models of component capabilities and interaction stories. In 3.4 and 3.5 we discuss the design of world model and controller. After that, we discuss three execution paradigms of interaction story in 3.6.

### 3.2 Inherent functionality: component capability

The functionality of a component defines its capability to serve the others, which determines the spectrum of behaviour that a component can perform.

#### 3.2.1 Basic model

A traditional and also widely used method to describe the behaviour of a system is through sequences of states or actions. A classical model is the Finite State Machine (FSM) model [122]. A typical example is the activity diagram [123], in which the states are action or sub-activity states and transitions are triggered by completion of the actions or sub-activities in the source states. An activity diagram captures the nature of system behaviour using dependencies between activities and is an important method to model software behaviour. In ISS-Conductor, the component capabilities are modelled using activity diagrams. The dependencies between activities are described using the execution states of activity performance, data, and condition guards. The quality descriptions of the activities are also included as part of the capability for the purpose of component selection and run-time performance adaptation. The component capability is thus defined as 5 elements: \((\text{Actions, States, Data, Transitions, Quality})\) where:

1. **Actions** is a set of activities that the component can perform. It always includes an initial action and a set of terminal actions. Each action is associated with two lists of shared data classes for indicating its input and output of data objects respectively. Before an action can be executed, the instances of all the shared classes in its input list have to be available in the input buffer.
3.2 Inherent functionality: component capability

2. **States** is a set of states that describe the possible execution status of the actions. It consists of two non-intersecting subsets: \( S_{\text{unfinished}} \), for describing the unfinished states, which always contains one initial state and a number of proceeding states, and \( S_{\text{finished}} \), for describing the finished states. At run time, the state of an action always starts from the initial state, then shifts to the proceeding states, and finally one of the final states.

3. **Data** is a set of typed data objects which can be either internal or sharable.

4. **Transitions** is a set of transitions between Actions. A transition is described using the state of the starting action, an event and a set of guard expressions. A transition is active when the guard expressions are evaluated to true and the instances of the shared classes described in the input list of the target action are available in the input buffer. An action is called *doable* when it has an active transition from the current action.

5. **Quality** specifies the quality attributes of the component activities and data. In Chapter 5, we will have more discussion on this point.

The capability can also be represented by an activity-transition graph. Assume we have a simulator for solving some equation. It has 5 actions \{Start, InitSimulation, DoStep, ExportResult, Stop\}. The action executions have four possible states: \( S_{\text{unfinished}} = \{\text{toDo, doing} \} \) and \( S_{\text{finished}} = \{\text{succeed, failed} \} \). The initial setting for the computational and control routines is represented in the data object *Setting*; the computational results are formalised as data object *Result*. The activity-transform graph is depicted in Fig. 3.2.

![Activity-Transition Graph](image)

**Figure 3.2:** A partial activity-transition graph of the example. In the description, actions: Start, DoStep and Stop do not have dependencies on data objects, the action InitSimulation requires a Setting object as its input, and the action ExportData has a Result object as its output. In the example, the action DoStep has a transition with a condition guard: \( \text{error} > \text{Setting.error} \), which means the action will only be performed when the error is larger than a given bound.

The ISS-Conductor is designed on top of object-oriented middleware. The issues related to the ownership of the objects such as object creation, destruction and update
will be handled by the services from the underlying software bus. If an object in the input class list of an action is not updated, the controller will request the owner of the object to update the content before executing the action. If an action has an object in its output list, and the component does not hold its ownership, then the action will issue a request to the software bus to negotiate the ownership before performing the update operations.

### 3.2.2 Capability modelling for the human interaction involved components

The run-time behaviour of a human-interaction involved component is influenced by both the MA and the human user, thus the human actions have to be considered when modelling its capability. Before discussing the details, we shall first take a look at the basic structure of a stand-alone interactive system.

From the perspective of task processing and human interaction, the functionality of an interactive visualisation system can be described as a hierarchical structure using the notions from Activity Theory \([124, 125]\)\(^1\). The top layer is a set of *tasks* supported by the system, the middle layer is a set of goal-directed *subtasks* which can be performed by the user to realise each *task*, and the bottom layer is a set of *operations* for carrying out each *subtask*. An *operation* can be mapped onto an element in the user interface. The interface manager of the interaction system ensures that the dependencies between tasks and interface elements are handled. Fig. 3.3 shows an example of an interactive visualisation tool which allows three basic *tasks*: selecting a task, exploring data and stopping the system.

---

**Figure 3.3: A simple model of human interaction involved systems.**

The capability model of an interactive system needs to meet two requirements, first, the run-time behaviour of the interactive system can be controllable by the activities in the capability, and second, it inherits the legacy support for human-centred

\(^1\)To avoid the unnecessary confusions with the capability model, we use the terms *task* and *subtask* instead of *activity* and *action* as the notions.
interaction. In principle, the activities of the system can be modelled at any level of tasks, subtasks and operations. However, it is not preferable to model the component activity at the operation level, because when the legacy controls for the dependencies between interface elements are taken by the Module Agent, the concurrency between the user activities and the agent control tends to introduce a large state space. Currently, we use the task level. Each task is modelled as two actions: enable and disable which influence the user interaction by enabling and disabling the interface elements respectively. The capability also includes actions for handling I/O operations. Although not being modelled as the actions in the component capability, the human actions are important in describing the dependencies between the agent actions. A suitable way is to model them as states: the subtasks that the user is performing are modelled as the state of the user behaviour. The other part of the capabilities, such as Events, Data, Transitions and Quality can be derived in the same way as for a non-interactive component. Fig. 3.4 shows the activity-transition graph of the Fig. 3.3.

![Activity-Transition Graph](image)

**Figure 3.4**: Capability modelling of the components involved with human interactions. A partial activity-transition graph of Fig. 3.3. The term InState describes the state of user activities.

### 3.3 Interaction: story and scenarios

A story provides rules to steer the run-time behaviour of the component instances. To decrease the complexity, a story is divided into a number of simpler fragments, called scenarios. Those scenarios can be reused in different stories. A scenario can be, in principle, specified in a number of ways, e.g. activity diagrams [126], state charts [127] and Petri Nets [128]. Because of the well-established theoretical framework and more importantly the suitability for representing the common flow patterns and concurrency dependencies [129], a Petri Net based approach [130] is adopted.

---

[126] Enabling and disabling interface elements is supported by most of the user interface development toolkits.
### 3.3.1 Place transition net

The concept of a Petri Net was originally developed by C. A. Petri in the 1960s. It has been a widely applied for modelling system behaviour, in particular concurrent activities. *Petri Net* has actually become a generic word referring to a body of research such as elementary net theories, place transition graphs, and high level graphs [131,132]. Place transition graphs are a sort of automata that can handle relations between conditions and the occurrence of events [130].

A place transition (PT) graph can be specified as a triple, $(SP, ST, SF)$ [133] where:

1. $SP$ is an finite set of places;
2. $ST$ is an finite set of transitions. $SP \cap ST = \emptyset$;
3. $SF$ is a map between $SP$ and $ST$, and between $ST$ and $SP$. $SF \subseteq (SP \times ST) \cup (ST \times SP)$.

A number of concepts are used to specify the properties of PT nets:

1. **Weight** is a map between $SF$ and natural numbers $\{1, 2, \ldots\}$. In this paper, we only use the nets that have equal weights, 1, for all links in the SF.
2. A **Marking** of a PT graph is a map between $SP$ and $\{0, 1, 2, \ldots\}$. $M(p)$ denotes the number that marks the place $p$, it also reflects the number of tokens in that place.
3. **Pre-set of** $x$, denoted as $\bullet x$, is defined as $\bullet x = \{ y \in SP \cup ST \mid (y, x) \in SF \}$, $x \in SP \cup ST$.
4. **Post-set of** $x$, denoted as $x \bullet$, is defined as $x \bullet = \{ y \in SP \cup ST \mid (x, y) \in SF \}$, $x \in SP \cup ST$.
5. A transition $T$ is **enabled** when each element of its pre-set has at least one token. $\forall p \in \bullet T$, $M(p) > 0$.
6. A enabled transition $A$ can be **executed**, also called **fired**. The execution will update the marking of the net. The update rule is that all elements in the pre-set of the transition decrease by one token, and all elements in the post-set of the transition increase by one token.
7. An execution of a PT graph can be described using the sequence of occurred markings. The set of all possible sequences yields a connected graph, named the marking graph of the PT graph.

Fig. 3.5 shows a model of data production-consumption relation. On the left side, a PT graph contains five places: $a$, $b$, $c$, $d$ and $e$, and four transitions: $t_1$, $t_2$, $t_3$ and $t_4$. The initial mark is $(1, 0, 0, 1, 1, 0)$ and $t_1$ is the first enabled transition. After executing $t_1$, the place $a$ passes one token to $b$; the mark of the net is updated as $(0, 1, 0, 1, 1, 0)$ and $t_2$ is another enabled transition. The marking graph shown on the right side of the figure describes all the possible marks.
### 3.3 Interaction: story and scenarios

#### 3.3.2 Scenario representation

A simple PT graph captures the qualitative properties between the concurrent activities, but not the quantitative properties of the conditions for a specific transition. A common solution is to associate additional information with either the nodes (places and transitions) or the relation links. In ISS-Conductor, the extension is added to places and the links between places and their post sets.

In a scenario, the activity dependencies between the roles are modelled using an extended PT graph, named a scenario net. It models the interactions using two types of dependencies between the activities of different component instances. The concurrency dependencies between them constitute the first type, which are represented as the relation links between places and transitions, and the tokens of the places. The second type of the dependencies are the specific conditions for each action, which are represented as the control expressions in the pre-set of the transition and in the links between the transition and its pre-set.

In a scenario net, transitions and places have unique names. Transitions are used to specify activities or nested scenario nets. When specifying an activity, a transition contains an action and a role name, where the role is expected to perform the action at run time. The role is called the responsible role of the transition. When specifying a nested scenario net, a transition contains the name of a scenario net and a special action called Do Scenario. Places and the links between places and their post sets are used to describe the conditions. A place is optionally associated with a set of expressions, named place expressions, which contain three subsets, for describing the initial conditions, control conditions and the state-modification rules of the place. Each link between a place and its post set is optionally associated a set of guard expressions. We will discuss the semantics of these expressions later. In the expressions, parameters are accessible by all roles when they are updated by the place expressions, or only accessible by a specific role when they are updated by the world model of that role. We use \( PR(\{P\}) \) to represent all the parameters that are read by the place expressions of
the place set \( \{P\} \) or the guard expressions in the links between all the elements in \( \{P\} \) and their post sets, and use \( PW(\{P\}) \) to represent all the parameters that will be updated by the place expressions of the place set \( \{P\} \).

The execution of a scenario net is dependent on its marking, the place expressions and the guard expressions in the links:

1. The initialisation of a scenario net takes two steps: it first assigns the initial marking \( M_0 \) to the places, and then executes the initial expressions in all places of the scenario net once.

2. A transition is enabled when all the places of its pre-set have at least one token, the control expressions in all these places are evaluated as \textit{true}, and the guard expressions in all the links between the transition and its pre set are evaluated as \textit{true}.

3. If the action in an enabled transition is doable, the transition can be executed (also called fired). The execution will update the marking of the scenario net. It first executes all the state-modification expressions of the places in the pre-set of the transition, and then updates the tokens of all the places in the pre-set and the post-set of the transition as in a normal PT graph.

The execution of a transition updates the state of the scenario net. The basic rules of the PT graph handle the concurrency dependencies between the activities by changing the marking of the net, and the execution rules for the place expressions update the control conditions for the activities. Since the responsible role of a transition will evaluate all the expressions in the places of its pre-set and in the relation links between them, it needs to have right to access the parameters in the expressions.

### 3.3.3 Transitions and actions

In a story four special actions, named 	extit{story-control actions}, are also defined: \textit{Start Scenario}, \textit{End Scenario}, \textit{Synchronisation Actions} and \textit{Do Scenario}.

1. In a scenario net, one and only one transition contains \textit{Start Scenario}. The transition defines the synchronisation point for starting the scenario net. The role that is responsible for doing the \textit{Start Scenario} action is the \textit{responsible role} for the scenario net.

2. In a scenario net, one and only one transition contains \textit{End Scenario}. The transition defines the synchronisation point for stopping the scenario execution.

3. A scenario net may contain a number of \textit{Synchronisation Actions}. The transitions define points for one or more roles to synchronise their activities.

4. A scenario net may contain a number of \textit{Do Scenario} actions. As we have mentioned, a \textit{Do Scenario} action defines the entrance to a nested scenario net. The responsible role of the nested scenario net is responsible for doing the \textit{Do Scenario} action.
The story-control actions are always doable. The actions specified in the scenario net must either be defined in the capability of a role or be story-control actions. In a scenario net, only the key activities of the involved roles need to be described, and the capabilities of the roles are responsible for searching intermediate actions to link them.

Let's take an example. There are two components a Producer and a Consumer, which capabilities are shown in Fig. 3.6. We want to build a simple scenario for three roles: Producer A is an instance of the component Producer, and Consumer A and Consumer B are two instances of the component Consumer. In the scenario, Producer A produces data for both Consumer A and Consumer B 10 times. Data transmissions between Producer A, Consumer A and Consumer B are through a software bus and use a publication/subscription mechanism, which means each data object produced by Producer A can be consumed by both Consumer A and Consumer B. The Producer A only continues when the Consumer A and Consumer B both finish their consumption (controlled by the Synchronisation Action at transition Sat5). In the scenario net, the place-expressions are only specified when they are not empty. The responsible role is Producer A. At its run-time, the parameters SA and SB increase by one after the Consumer A and Consumer B consumed a data. The expressions in Sap2 and Sap7 control the branch after the Sat5.

![Figure 3.6: A sample scenario for roles Producer A, Consumer A and Consumer B. More examples will be discussed in the next chapter.](image)

3.3.4 Story: a scenario-net instance

A story is a scenario-net instance. It may contain a number of nested scenario nets which are also called scenarios of the story. When the End Scenario action of a story has been executed, the system will exit. The responsible role for doing the End Scenario action broadcasts an exit message to all the roles in the system, and
the peer roles will do the actions that are in a path leading to one of the \emph{terminal actions} in their capabilities.

\section*{3.4 World model}

To behave rationally in a story, a role has to know not only its own execution status but also the progress of the other roles. Each role has to track the state of the entire system in order to make correct decisions on its activities. The \emph{world model} provides the necessary services.

\subsection*{3.4.1 Basic structure}

The world model tracks and processes the changes of the external world using a uniform structure \{\emph{parameter, observations, perception}\}. \emph{Parameters} are the things that are being tracked and \emph{observations} are the temporary value of parameters, which are ordered by their time stamps. The perception analyses the observations, both the value and their time stamps, and maps them to a set of qualitative descriptions or obtains the latest value, called \emph{belief} of the parameter. Based on the type of information, parameters are classified in five groups:

1. \emph{Agent world} related parameters are the names of the involved roles in the story. The perceive function returns the believed states for the role. The state of a role is determined based on two issues: if the role is present in the system, and if the role has recently updated its state. Four states are defined: \emph{never heard of}, \emph{is updated}, \emph{is not updated} and \emph{has disappeared}. The semantics and the transitions between them will be discussed later.

2. \emph{Story} related parameters are for scenario changes. The perception function returns its believed story state, which includes the current scenario and the state of the scenario-transition graph.

3. \emph{Scenario} related parameters are for the marking of the scenario net. The perceive function returns the believed marking for the scenario.

4. \emph{Execution} related parameters are for activities and their states. The perceive function returns the believed current action and its state.

5. \emph{Data} related parameters are the names of data objects. The perceive function returns the believed value of object attributes.

\subsection*{3.4.2 Perception and uncertain belief of the agent world}

For the agent world, \emph{perception} can use the value of the observations as well as statistical functions, e.g. minimum, maximum and average of the intervals between the time stamps of the observations, and a \emph{belief-transition} graph to derive the belief
of a parameter. A belief-transition graph is a state machine based model, in which the states describe the possible belief and the guards in the state transitions are described using the statistical functions or the values of the observation. Initially a role perceives all the peer roles as *never heard of*; after receiving state-update messages, the beliefs of the corresponding peer roles will be turned into *is updated* or *is not updated* depending on the time intervals for updating new states; finally, if the role has not received any messages from a peer role for a relatively long interval, the peer role will be perceived as *has disappeared*.

In the belief-transition graph, not all the transitions can easily be represented using a single function, e.g. an agent can not distinguish whether a peer role is *not updated* or *has disappeared* when no messages have been received from that role for a period of time. Based on the fuzzy state machines discussed in [134, 135], uncertain belief is introduced to reason on those situations, as shown in Fig. 3.7. Two types of belief are defined: *certain* or *uncertain*. The *uncertain* beliefs are associated with a set of operations, called *proof actions*, which can be invoked for gathering additional information to make the belief certain. The degree of the *uncertainty* is represented by a real number which is between 0 and 1. If a belief is certain, its degree is always 1. When it is an *uncertain* belief, the invocation of the proof actions will change the degree of the uncertainty. When the degree of the uncertainty achieves zero or one, the belief will be transferred to a *certain* one. The world model can hold an incorrect belief about the neighbours, e.g. when the network connection temporarily breaks, the neighbours will be perceived as *has disappeared*, but after the connection resumes, the belief will be turned into *is updated*, as shown in the graph.

*Figure 3.7: A belief-transition graph for deriving the states of neighbour roles.*
3.5 Controller

The controller co-ordinates the functional components in an MA and collaborates with the other roles to carry out the story execution. In more detail, the controller processes the information observed by the sensor, updates the world model, finds suitable actions from the story and capability, and controls the execution of the actions.

3.5.1 Collecting observations

The first thing that the controller does is to collect the information observed by the sensors. The sensors are actually the ComA of the Conductor. There are basically three types of information which could be observed by the ComA. The first one is signals that the software bus passes to the ComAs, which are normally generated by the protocols of the underlying middleware. The second one is messages that components send to each other. The purpose of a message is either to update or to query the state information. The difference between them is that the second type of messages expects a reaction from the receiver. The third one is the reflection of the new value of data objects and their attributes.

The ComA in the Conductor observes the events from the external world and passes them to the controller in the MA. The controller then generates more specific events for the world model to update corresponding parameters. The controller checks regularly whether the world model has any proof actions that need to be invoked.

3.5.2 Action execution control

The controller also controls the action execution. In general, the controller handles two types of actions: those specified in the story or the capability, and those that are a response to the normal events, including proof actions. The first type of actions can only be executed when the previous one has been finished, while the second one can be executed at any time. For the first type of actions, if it is not a story control action, it will have to be sent to the Actor. The controller handles the protocols for action sending and searching using the states provided by the world model, as shown in Fig. 3.8.

![Figure 3.8: The action control between an Actor and a conductor, and its reflection in the world model.](image-url)
The action requested by the story is possibly not immediately doable by the capability, but can be doable after a certain number of intermediate actions. For this kind of actions, the controller considers the states of the action that is executed by the Actor as a nested state, and uses it to control the update of the story state, as shown in Fig. 3.9.

Another main function of the controller is to collaborate with the other roles to execute a story. This will be discussed in the next section.

### 3.6 Story execution

At run time, the MA in a component interprets the story and controls its behaviour. The MAs collaboratively orchestrate the overall system behaviour in three possible paradigms: distributed, hierarchical and centralised.

#### 3.6.1 Basic paradigm: distributed scenario execution

In *distributed scenario execution* each role maintains its own execution state of the story, and independently finds enabled transitions from the scenario net using its local states. Execution proceeds through four basic phases: finding actions, executing actions, updating the local story state and synchronising the state with the other roles.

**Finding actions**

Searching for an action normally takes two steps. The MA first finds an enabled transition from the scenario net, and then checks if the action defined in the transition is doable. A story-control action is always doable and a normal action has to be approved by the capability. When the action in the enabled transition is not directly
doable, the searching rules will find an intermediate action from the capability, which leads a path to it.

**Action execution and concurrency control**

Story-control actions are executed by the Conductors, and the normal actions are executed by the Actors. Before an action is executed, its safety with regard to possible concurrently executed actions has to be checked. A concurrency conflict occurs when there are two transitions for which two different roles are responsible, and the execution of one transition might disable the condition of the other one. Fig. 3.10 shows an example of concurrency conflicts and the possible illegal markings. The scenario fragment contains four transitions: role A is responsible for T1 and T3, and role B is responsible for T2 and T4. When executing in the distributed paradigm, both role A and B have an initial mark of the scenario net: (1, 0, 0, 0), and both of them have an enabled transition: T1 and T2 respectively. If both of them simultaneously execute the transitions, the mark of the scenario net will be turned into (0, 1, 1, 0), which is apparently invalid in the actual marking graph of the scenario net, as shown on the left bottom of the figure. In the scenario net, T1 and T2 have concurrency dependencies, and are called critical transitions.

![Diagram](image)

**Figure 3.10:** A scenario fragment and its marking graph are shown on the left side. The right side shows a possible execution sequence of role A and B, when they do not apply any concurrency controls. The dashed arrows indicate the marking changes that are perceived by the peer role; the markings in Italic font are invalid.

The concurrency conflicts can be checked using the following rules. Assume that role A and B are responsible for two different transitions $T_i$ and $T_j$ in a scenario $S$. The transition $T_i$ and $T_j$ are critical transitions when executing either of them might
change the condition of the other: the number of tokens in the pre-set or the condition expressions in the places and in the relation links. In more detail, the critical transitions can be identified using \( \bullet T_i \cap \bullet T_j \neq \emptyset \) or \( \text{PW}(\bullet T_i) \cap \text{PW}(\bullet T_j) \neq \emptyset \) or \( \text{PR}(\bullet T_i) \cap \text{PW}(\bullet T_j) \neq \emptyset \) or \( \text{PW}(\bullet T_j) \cap \text{PR}(\bullet T_j) \neq \emptyset \). Role R is an involved role for a critical transition \( T_i \) when R is responsible for \( T_i \), or it is responsible for a transition \( T_j \) which is in conflict with \( T_i \).

To execute a critical transition, a role has to negotiate with the other involved roles to ensure mutual agreement. A negotiation is designed based on the algorithm described in [136]. The local time of the roles is used for the comparison; the one with the smallest value wins.

### State update and synchronisation

After a transition has been executed, a role first updates its local state of the scenario net and then causes the other roles to synchronise their local states. A role broadcasts an update-request message to the other roles when it has updated its local state. The peer roles synchronise their local states by mimicking the execution of the state after receiving the state update request message. The synchronisation operation can only take place once for each request.

The services provided by the Run Time Infrastructure of HLA, e.g. ordered-message delivery, can not easily handle the synchronisation, because message delivery is not always reliable, e.g. due to a temporary loss of the connections between roles. Processing the messages in a Receive-Ordered (RO) way, a role cannot ensure that all its requests for state-update have been received by the peer roles, neither can it ensure that it has received all the requests from the peer roles. On the other hand, with Time-Stamp-Ordered (TSO) message delivery, the grant of the federation time will be blocked by one agent when it is out of function. A high-level control for the synchronisation is needed.

To record the executions of transitions, each role maintains two groups of data structures in its world model. The first one is called master table, which records all the transitions that it has executed, and each item in the master table is also associated with an acknowledgement table. And the second one is called slave table which records all the transitions for which update request messages were received; a separate slave table is maintained for each peer role. Fig. 3.11 shows the basic structure.

These tables ensure that any executions performed by a role will be synchronised by all the peer roles exactly once. First, the sequence of the transitions executed by a role is tracked using the master table, the acknowledgement tables check if the transitions have been reflected by all the peer roles. Second, the slave tables record the histories of the transitions executed by the peer roles, which guarantees that the local update for each transition only takes place once.
3.6.2 Hierarchical execution paradigm

In the distributed paradigm, the state update messages are broadcasted. If the number of roles or transitions increases, the number of messages will also dramatically increase, and the massive number of small size messages will degrade the system performance. The hierarchical execution paradigm is proposed to overcome this problem. It intends to limit the number of messages by only sending the update requests to the roles that really need them. First we will define some basic concepts.

We define that two roles $R_a$ and $R_b$ are **tightly dependent**, denoted as $tDep(R_a, R_b)$, when $T_a$ and $T_b$ which are the responsibility of role $R_a$ and $R_b$ respectively satisfy at least one of the following conditions: $(\bullet T_a \cap T_b \bullet) \cup (\bullet T_b \cap T_a \bullet) \cup (\bullet T_a \cap \bullet T_b) \neq \emptyset$ or $(PW(\bullet T_a) \cap PW(\bullet T_b)) \cup (PW(\bullet T_b) \cap PR(\bullet T_a)) \cup (PR(\bullet T_a) \cap PW(\bullet T_b)) \neq \emptyset$. Two roles $R_a$ and $R_b$ are **loosely dependent**, denoted as $lDep(R_a, R_b)$, when $T_a$ and $T_b$ are not tightly dependent, but there exists a sequence of roles $\{R_1, R_2, \ldots, R_n\}$ which has $tDep(R_{a_1}, R_{b_1}), tDep(R_{a_2}, R_{b_2}), \ldots, tDep(R_{a_n}, R_{b_n})$. And two roles are **dependent** when they are either tightly or loosely dependent. From the definition, we can see if a scenario net is connected, which means no transitions are isolated from the others, any two roles are dependent.

The basic idea of the hierarchical paradigm is that all the roles which are tightly dependent should receive the update request messages from each other. Therefore, the distribution of the update-request messages will be multicast instead of broadcast. To construct the multicast groups, a **dependency tree** is proposed:

1. The nodes in the tree contain a non-empty set of roles, the root node only contains one role that is the responsible role for the scenario. A role can only belong to one node in the tree. The $\{R\}$ is the set of all the roles in the node.

2. For a role $R$, all the roles that are tightly dependent with it must belong to one of the possible nodes: the same node with $R$, its parent node, or one of its child nodes.
3. Any two roles $R_a$ and $R_b$ which belong to the same node must be either tightly dependent or loosely dependent. And if they are loosely dependent, there must exist a set of roles $\{R_s\} = \{R_{s1}, R_{s2}, \ldots, R_{sm}\}$ which has $\{R_s\} \subseteq \{R\}$ and $tDep(R_a, R_{s1}), tDep(R_{s1}, R_{s2}), tDep(R_{s2}, R_{s3}), \ldots, tDep(R_{sm}, R_b)$.

4. Any two roles $R_a$ and $R_b$ which neither belong to the same node, nor to parent-child node pair must not be tightly dependent.

In a connected scenario net, any two roles are dependent, which means a dependency tree can always be derived. Using the dependency tree the multicast groups for distributing state-update messages can then be allocated. All the roles that are in the same node or the parent-child nodes will be in one group, and the roles that do not belong to the same node or parent-child nodes will be in different group. The multicast groups are handled using the data distribution services provided by the underlying middleware, which will be discussed in the next chapter.

### 3.6.3 Centralised coordinator paradigm

The final execution paradigm is to interpret a scenario using a centralised co-ordinator. An important reason for employing this paradigm is the cost of the negotiation operations, especially when the number of critical transitions and the number of their involved roles are large. The centralised co-ordinator paradigm works as follows:

1. The responsible role of the scenario is set as the co-ordinator.

2. Only the co-ordinator maintains the state of the scenario net, and only the co-ordinator searches the enabled transitions.

3. The co-ordinator searches enabled transitions not only for itself but also for all the other roles in the scenario.

4. When a subordinate role receives a transition sent by the co-ordinator, it will check the doable actions using its own capability. When the action in the transition has been executed, it sends back the execution state to the co-ordinator, and requests for the next enabled transition.

5. Between subordinate roles, no update request messages are sent.

To allow the co-ordinator to check the enabled transition for different roles, the parameters defined in all place expressions should be accessible by the co-ordinator. Therefore, for this paradigm, the subordinate roles necessarily send the value of their private parameters to the co-ordinator during the scenario execution.
3.6.4 Scenario switch and execution paradigm selection

When entering an ISS, a role initialises the story as the first scenario. A new scenario is switched on when a Do.Scenario action is executed. The role that executes the Do.Scenario action first broadcasts a scenario-switch message to all the roles, then saves the execution state of the current scenario, which includes the mark of the current scenario net and the setting of its execution paradigm, after that it initialises the new scenario. When a peer role receives a switch announcement, it saves the state of the current scenario, and then initialises the new scenario. Currently, no parallel scenarios are allowed, which means at one time, there can only be one active scenario in the system.

When a scenario is started, its responsible role makes a decision on the execution paradigm, which is based on two basic facts: the total number of the involved roles and the total number of critical transitions in the scenario. The facts for designing the rule are that the centralised paradigm can handle the critical transitions more efficiently than the other two paradigms, but its performance will decrease when the total number of the roles is large. The threshold values for the number of roles and the number of transitions are empirically set in the knowledge base. The responsible role announces the decision on the execution paradigm to the other involved roles before executing the action Start.Scenario.

Roles choose multicast groups for receiving messages and data objects when entering a scenario. If a role is not involved in a scenario, it will not join any groups for receiving state-update messages. But the messages for scenario switches are broadcasted to all roles. In the next chapter, we will discuss how the adaptation is realised using the distribution routing spaces.

When a role executes the End.Scenario action, the current scenario will exit. If the current scenario is the story, the execution of the system will finish. Otherwise, the role restores the state of the previous scenario, then broadcasts a restore message to all the other roles. Exiting a nested scenario means the completion of a Do.Scenario action in the previous scenario, the role therefore also needs to update the states of the previous scenario net and announces a state update message to the other roles. The history of the scenarios is tracked by the master and slave tables of the story.

3.6.5 Handling run-time exceptions

At runtime, the execution of a story can have a number of exceptional situations, e.g. some roles never show up in the scenario, join the scenario while the others have already started the execution, or suddenly disappear from the system. These exceptions can have different reasons such as the temporal loss of the network connection, a temporarily unavailable computational infrastructure, or internal errors in the component, which can occur at any time, especially when the run-time environment contains a large collection of heterogeneous computational resources such as computational Grids. To execute a story robustly, the MA employs a number of strategies to handle those exceptional circumstances.
First of all, a scenario can only be started when the responsible role is present. This implies two things: when a story is started, the responsible role for the story has to be present, and a Do Scenario action can only be executed by the role that is responsible for the nested scenario net. The presence of a role is determined by the world model. Secondly, a role is not allowed to enter into a scenario when the other roles have already started the execution. It means that the normal roles of the scenario have to be present in the scenario before the responsible role starts the execution. This strategy is also related to a third one that the transitions that are the responsibility of absent roles will always be considered as enabled. Making this assumption avoids that the execution is blocked by an absent role. During the execution, if a component crashes, the other roles will eventually perceive that role as disappeared from the story. If a role disappeared from the story, the other roles will use the same strategy as for an absent role to handle the rest of the scenario. If the responsible role of the scenario crashes, the roles will switch their scenario to End Story to terminate the system. Finally, if the story contains a invalid transition, e.g. the action in the transition is not defined in the capability or will never be doable from the current state, the responsible role of the transition will announce a message to the other role that it quits from the scenario. And the other roles will use the second and third strategies to handle the rest of the scenario.

3.7 Summary

In this chapter, we have discussed the core design of Module Agents and the mechanisms for orchestrating interaction among them. As we have discussed in the previous chapter, ISS-Conductor realises the separation between system functionality and application specific interactions using a layered agent framework, in which ComAs realise the basic communication details and MAs provide an abstract structure to control the activity. The component functionality is modelled as a finite state machine (capability), which can be programmed with the other components using a Petri net based mechanism (scenario net). In this chapter, we have not presented the experimental results of the development, but from the discussion, we can enumerate a number of design characteristics.

1. Describing the capabilities of the components and the interaction dependencies separately is essential to support scientists to prototype an ISS from high level.

2. The capability of a component is modelled using a Finite State Machine based model. A uniform mechanism is proposed for describing the capabilities of both normal components and components involved human-interaction. In the model, the human activities are modelled as different states based on the activity theory.

3. The interaction dependencies between components are modelled using a Petri net based mechanism, called scenario net. In a scenario net, component activities are described in transitions, and conditions for the activities are described
using expressions in places and in the relation links. The rich semantics of Petri nets can describe the interaction constraints not only from the perspective of data dependencies, as often used in scientific workflow systems, e.g. SciRun, Sculf, and GridAnt [16, 137, 138], but also from the concurrency relations between activities. It achieves a paradigm for rapid prototyping of ISSs.

4. The world model plays an important role in the run-time control of system behaviour. In ISS-Conductor, the world model includes fuzzy states in the perception of the other MAs.

5. Three execution paradigms are proposed in MAs. The execution of ISS-Conductor offers more flexible paradigms than centralised control: distributed and hierarchical ones.

In the coming two chapters, we will first discuss the implementation details of ISS-Conductor and then use a medical application as a test to demonstrate the main features of the architecture.