Design Patterns for and Automation of Federation State Control

T.W. van den Berg
R.E.J. Jansen
TNO Defence, Security and Safety
PO Box 96864,
2509JG The Hague, The Netherlands
tom.vandenberg@tno.nl, roger.jansen@tno.nl

H. Ufer
IABG
VG14
Schießplatz (Gebäude 170)
PO Box 17 64
D-49707 Meppen, Germany
ufer@iabg.de

Keywords:
State Control, Design patterns, Federation Agreements, XML, HLA.

ABSTRACT: *The suitability of or choice for a particular design pattern for federation execution state control depends on several factors, such as the degree in which legacy applications can support particular patterns and the complexity of the federation. For small federations that involve only a handful of federates that execute at one location, it may be sufficient to use manual control of the applications and document all necessary procedures in a Federation Agreements document, whereas a complex federation executing on a wide area network can find many benefits in using automated execution state control.*
Standaridisation of design patterns for federation execution state control will promote re-use of federates in different federation configurations. And formalisation of these design patterns using available standards will enable a certain degree of automation of execution state control, independent of the federation.
This paper describes the functions of Federation Management, in particular execution state control, and provides an overview of a number of commonly found execution state control patterns. It introduces the W3C State Chart XML (SCXML) as state machine notation and demonstrates its use in the automation of federation execution state control using available open source software from the Apache Commons project.

1. Introduction

The concept of design pattern originates from Christopher Alexander back in 1977. In [1] he describes: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Erich Gamma et al applied the concept to software and published in 1995 the book “Design patterns: Elements of Reusable Object-Oriented Software” [2]. This book describes what design patterns are by providing amongst others standard elements associated with patterns and many examples.

The concept of design pattern can also be applied to execution state control for an HLA federation. Typical design issues like federation initialization and termination; start, stop, pause and resume; iteration; replay; etc; appear over and over in every HLA federation. The identification of common state control design patterns will benefit federate developers and further stimulate the reuse of federates in different federation configurations.

State and activity diagrams are common practice for modelling the behavior and functions of a system and are a logical way to express execution state control design patterns. With the widespread use of the Unified Modeling Language (UML) also the use of Harel statecharts has increased. Harel statecharts are in effect state diagrams, extended with the notion of hierarchy, concurrency and communication [3], [4]. A recent development is that of SCXML (State Chart XML) of the W3C [5]. SCXML is an XML [6] based markup language for the definition of state machines. The language is based on Harel statecharts and UML and it supports execution by a so called execution environment. Although SCXML is still a working draft it appears to be a suitable candidate for state machine representation and automated execution for federation state control.

A Community of Practice on Federation Architecture and Design established in the frame of the Technical Activity Program MSG-052 of the NATO Modelling and Simulation Group (NMSG) has elaborated and published proposals for common execution state control patterns for (geographically) distributed simulations [12]. The suitability of these proposals is presently being investigated in an experimentation program of the NMSG Technical Activity Program MSG-068 (NATO Education and Training Network (NETN)). The experiences gained in these experiments will form the basis for recommendations pertaining to federation execution control in the NETN Federation Agreement documents (FADs). The authors of this paper are members of both MSG-052 and MSG-068.

The remainder of this paper is structured as follows:

- First follows in chapter 2 a brief introduction into the basics of state and activity diagrams.
- Chapter 3 defines the general structure of an HLA federation explaining the role of the Execution Manager federate and Participating federates.
- Chapter 4 discusses a proposed template for a state control pattern for the Participating federates.
- Chapter 5 describes a number of state control patterns for the Execution Manager federate.
- Chapter 6 introduces the W3C State Chart XML as state machine notation for state control patterns.
- Chapter 7 describes an Execution Manager federate that is capable to execute SCXML and that is constructed with open source software components.
- Finally, chapter 8 summarizes the conclusions and further developments.

2. State Diagrams and Activity Diagrams

Many readers will be familiar with state diagrams for modeling the behavior of a system and activity diagrams for modeling the functions or capabilities of a system and how these two diagrams are related. This chapter briefly describes the most important elements of these diagrams in order to provide the reader enough information to understand the state control patterns described later in this paper.

2.1 State Diagrams

Figure 1 shows a simple state diagram: a composite state A that is decomposed in two sub states, named state 1 and state 2. In this example state 1 is the initial state and state 2 is the final state of the state machine. A transition from state 1 to state 2 is initiated by an event, called 'trigger' (in MSG-052 terminology State Transition Request (STR)), under the condition that the 'guard' evaluates to 'true'. A description of the effect of a transition is shown behind the forward slash.

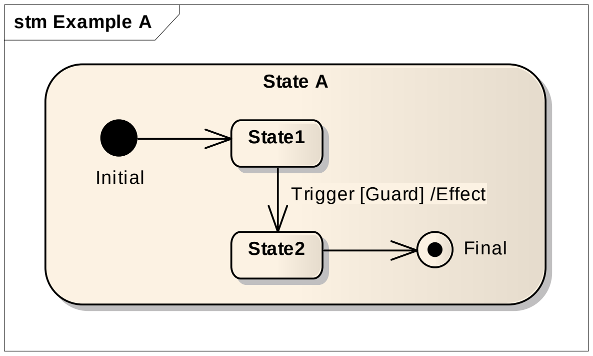


Figure 1: A state transition.

The states shown in Figure 1 are so called "or-states", that is, the system is in exactly one of the shown states. States can also be "and-states", that is, a system can be in multiple states at the same time. Figure 2 shows the notation for and-states. State B is split in two regions by a dashed line and each region contains a state diagram consisting of or-states. The system is both in a state from region B1 as in a state from region B2. And-states are also called "concurrent states".

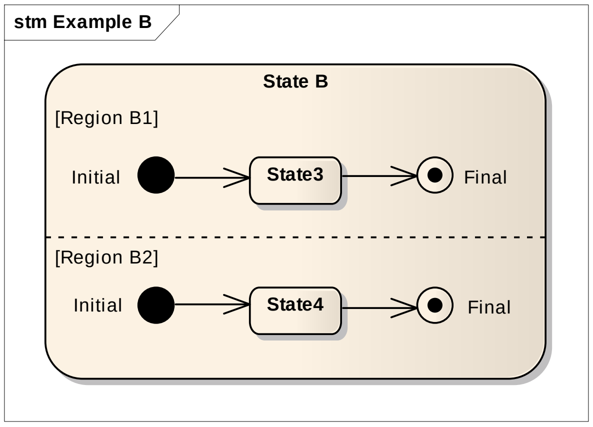


Figure 2: Concurrent states.

Pseudo states are an abstraction for various types of transitions used in state diagrams, the most common ones being the initial state and the final state. Another pseudo state is called 'junction'. A junction can be used to split a transition into multiple transitions, each with possible guards, as shown in Figure 3.

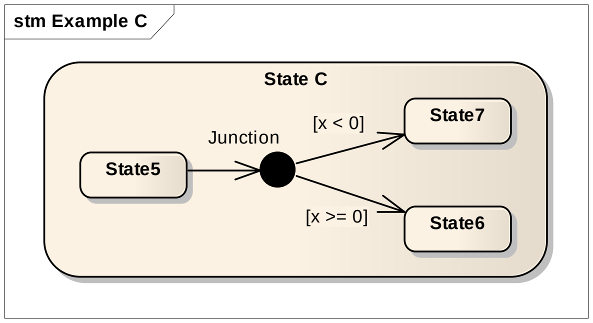


Figure 3: Pseudo state: junction.

2.2 Activity Diagrams

A trigger in a state diagram may imply certain activities to accomplish the desired effect. For each trigger there may be a related activity diagram. The name of the effect in the state diagram corresponds to the name of the activity in the activity diagram.

In the state control patterns described later in this paper each activity (i.e. effect of a transition) is named as "Do*Name*". An activity may consist of sub activities or actions, where an action is a basic functional unit of work and cannot be further subdivided. Figure 4 provides an example where on the transition from state 8 to state 9 the activity "DoSomething" is performed. The activity is shown in Figure 5 and consists of two actions.

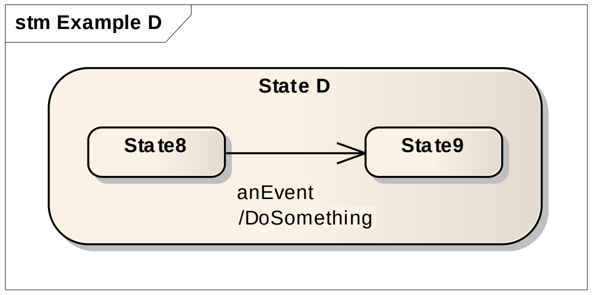


Figure 4: State diagram with reference to activity.

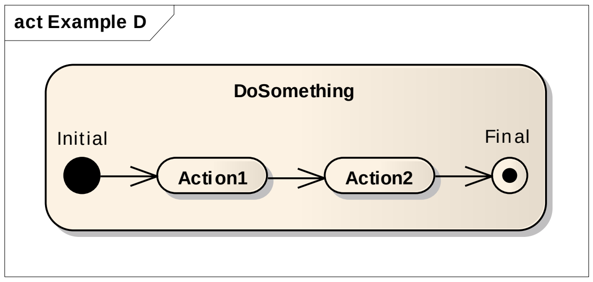


Figure 5: Activity referenced from state diagram.

3. Federation Structure

Before we describe the state control patterns in the next two chapters we first define in Figure 6 the general structure of an HLA federation. We assume that the execution management functions are centralized; distributed execution management is not in the scope of this paper. Consequently, a federation consists of an Execution Manager (EM) federate and one or more participating federates, interconnected via the Run Time Infrastructure (RTI).

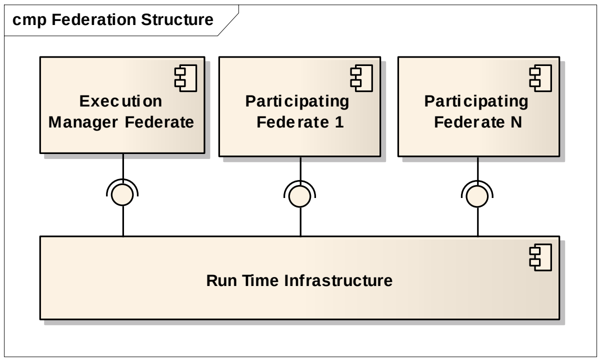


Figure 6: Federation structure.

The EM federate controls the federation execution. It initiates the transitions between states using the HLA Federation Management services (synchronization points and save/restore points) and Simulation Management interactions. The EM federate may have a GUI to initiate certain execution management events, such as start or stop the simulation. The participating federates are so called execution managed federates. They are controlled by the EM federate and receive state transition requests via the HLA Federation Management services or as FOM specific Simulation Management interactions. They perform the actions as required for each state.

To implement a state control pattern the following components should be in place:

- Federation Agreements that refer to the agreed state control pattern, describe the triggers, conditions and related activities of relevance to federation execution control;
- An EM federate that controls the federation execution according to the execution management state control pattern;
- Participating federates that are controlled by the EM federate and execute according to the participating federate state control pattern.

4. Participating Federate State Control Pattern

One of the outcomes of MSG-052 is a proposed template for a state control pattern for participating federates as discussed in this chapter. This pattern is expected to cover a large set of federate applications. However it is recognized that additional patterns may be required for certain application areas, such as for stochastic simulation, as discussed in chapter 5.

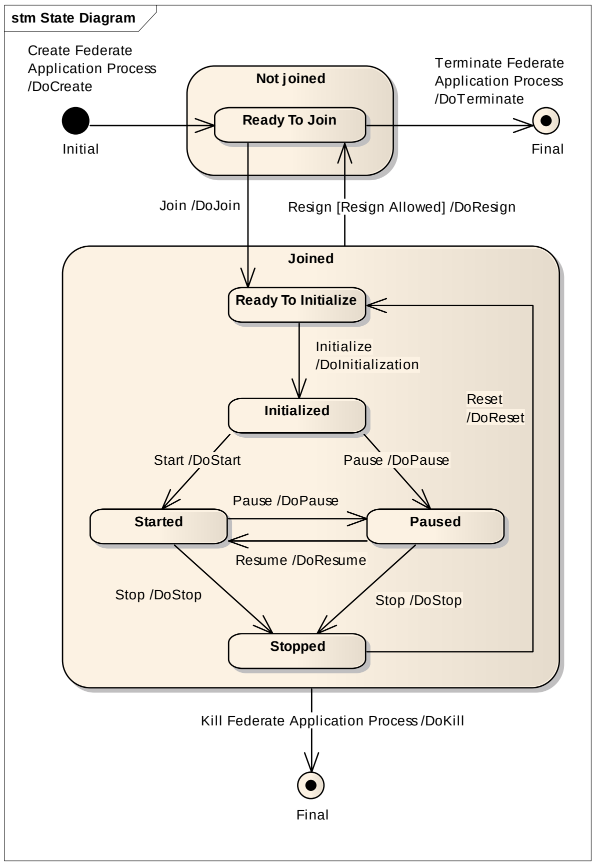


Figure 7: Participating federate state control pattern.

The states are defined as follows:

- **Not Joined:** In this major state, the application is not a member of the federation execution.

- **Ready To Join:** The application completed some internal set-up actions and is now ready to join the federation execution.

- **Joined:** In this major state, the application is a member of the federation execution.

- **Ready To Initialize:** The federate is neither time regulating nor time constrained, it has neither published nor subscribed any object or interaction class.

- **Initialized:** The federate may be time regulating and time constrained, it has completed its initial publications and subscriptions, it has registered its initial objects, it has sent initial attribute values for all registered objects, it has discovered all initial object instances from other federates and it has received all initial attribute value updates.

- **Started:** The federate is advancing simulation time.

- **Paused:** The federate does not advance in simulation time. The federate must be able to return to the Started state.

- **Stopped:** The federate does not advance in simulation time.

This pattern defines the major states, activities, and the names of the events. However, it neither contains a detailed description of the activity related to a state transition, nor does it imply which events trigger a state transition. These details are federation specific and must be defined in the Federation Agreements document. For example, an event can be triggered manually, by interactions or by the federation management services (synchronization points, save/restore points) or by other means of inter-process communication. The activity DoInitialization may include for example the following actions to be described in the Federation Agreements:

1. Enabled time management, if required.
2. Publish and subscribe to initial object and interaction classes.
3. Register initial object instances with the federation execution and update the attributes with initial values.
4. Achieve synchronization point.

To perform an activity on a transition the definition of a sub state diagram may be needed. For example if DoInitialization requires attribute values of initial object instances from other federates to complete the initialization then the activity has to wait on discover object and reflect attribute value events from the RTI. In this case additional sub states are needed to handle this, such as sub states like ObjectsRegistered, ObjectsUpdated and ObjectsDiscovered. Once the activity reaches the final state of this sub state diagram it is possible to transition to the next state, Initialized. These sub state diagrams result in additional patterns for performing specific activities, further refining the template pattern in Figure 7.

The pattern in Figure 7 only shows successful state transitions. If a state transition fails then it should be possible, depending on whether the problem can be fixed for a retry or not, to return to the originating state or to go to an error state. A more complete pattern should also includes these state transition failures and possibly an error state, that only allows to resign, terminate or kill the federate.

We hope that many specific federation execution management patterns can be aligned with this general state diagram. In that case, federates implementing this state diagram will be able to participate in many federations with different federation execution management patterns. The next chapter demonstrates how specific state control patterns can be aligned with this federate state diagram.

5. Execution Manager Federate State Control Patterns

This chapter describes four execution management state control patterns for controlling participating federates: three Start-Stop patterns for real time simulation and an Iteration pattern for non real time simulation.
The Start-Stop patterns use simulation management interactions and HLA synchronization points for execution control. The first Start-Stop pattern is very simple. As more features are added to each successive Start-Stop pattern the complexity of the pattern increases.
The Iteration pattern uses HLA synchronization and save/restore points for execution control.

In summary the patterns are:

Pattern name	**Simulation**
1. Start-Stop	Real time
2. Start-Stop with initialization	Real time
3. Start-Stop with initialization/reset	Real time
4. Iteration	Non real time

5.1 Start-Stop Pattern

The Start-Stop pattern is illustrated in Figure 8 and is a simple state control pattern where we use Simulation Management interactions to start, pause or stop the time advancement of the simulation. This pattern is for participating federates that – from execution management point of view – only support the Paused and Started states.

The EM federate has two states:

- **Waiting**: EM federate is waiting for the Start or Resign event. Participating federates are in the Paused state.

- **Running**: EM federate is waiting for the Pause or Resign event. Participating federates are in the Started state.

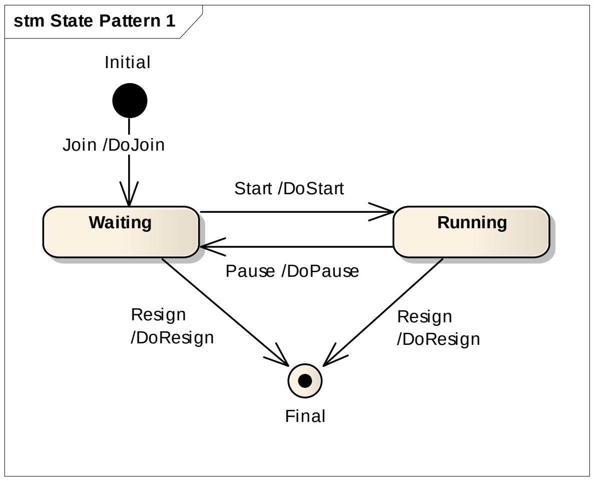


Figure 8: Start-Stop state diagram.

On the Join event the EM federate creates and joins the federation execution and enters the **Waiting** state. Participating federates are expected to cycle individually through the Ready To Initialize and Initialized states and perform the activities join, initialize and pause. Each participating federate ends in the Paused state and waits for the StartResume simulation interaction from the EM federate.
On the Start event the EM federate sends the StartResume simulation interaction and transitions to the **Running** state. Upon receipt of the interaction the participating federates are expected to transition to the Started state and start the time advancement of the simulation.
On the Pause event the EM federate sends the StopFreeze simulation interaction and transitions back to the **Waiting** state. Upon receipt of the interaction the participating federates are expected to transition to the Paused state and stop the time advancement of the simulation
On the Resign event the EM federate sends a StopFreeze interaction. It resigns from and destroys the federation execution, and terminates. Upon receipt of the interaction the participating federates are expected to stop the time advancement of the simulation, resign from the federation execution and terminate.

The Join, Start, Freeze and Resign events trigger the state transitions and these events are typically generated from the EM federate user interface.

The EM federate activities and actions are summarized in Figure 9. To preserve space in the activity diagram the control flow arrows between the actions have been omitted.

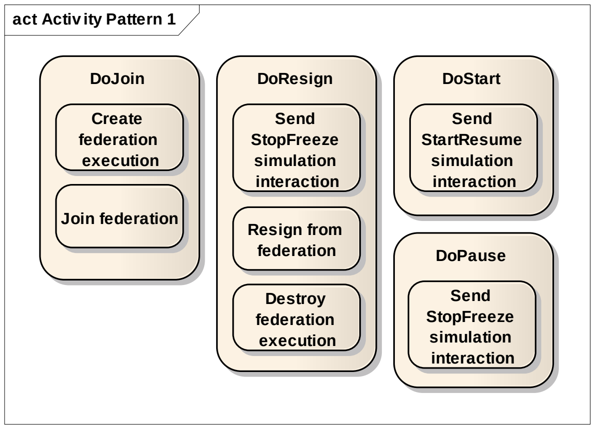

act Activity Pattern 1
DoJoin
Create federation execution
Join federation
DoResign
Send StopFreeze simulation interaction
Resign from federation
Destroy federation execution
DoStart
Send StartResume simulation interaction
DoPause
Send StopFreeze simulation interaction

Figure 9: Start-Stop activity diagram.

5.2 Start-Stop with Initialization Pattern

Often some kind of initialization is required amongst the participating federates. For example, time management needs to be enabled, initial object instances need to be registered and updated. With respect to the Start-Stop state diagram a number of additional states are introduced to support this (and more) from execution management point of view:

- **Joining**: EM federate is waiting for the participating federates to join the federation execution.
- **Initializing**: EM federate is waiting for the participating federates to initialize.
- **Suspended**: EM federate is waiting for the Start or Pause event. Participating federates are in the Initialized state.
- **Stopping**: EM federate is waiting for the participating federates to stop.

The state control diagram is shown in Figure 10.

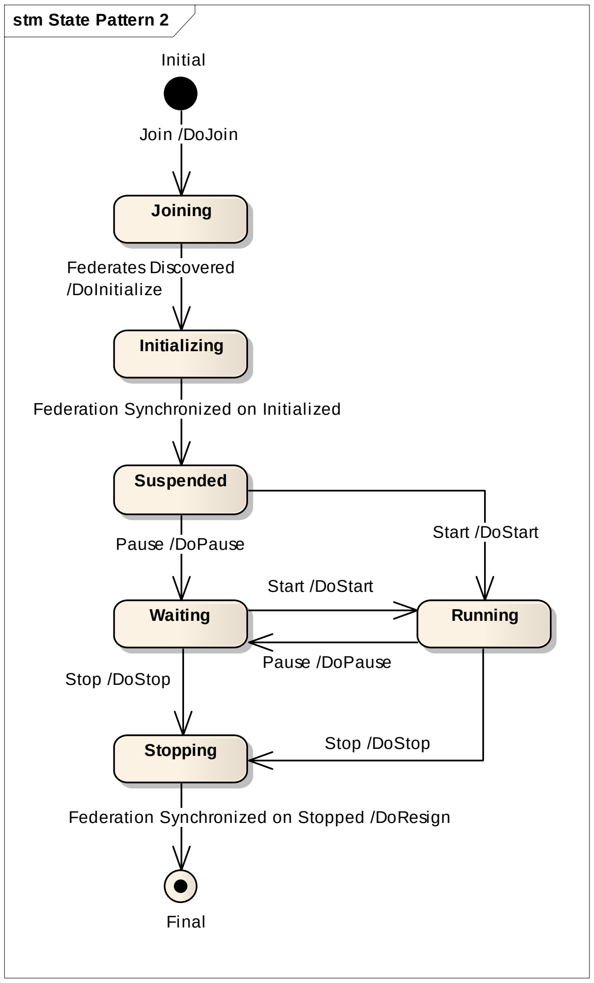


Figure 10: Start-Stop with initialization state diagram.

On the Join event the EM federate creates and joins the federation execution and waits for the other participating federates to join and transition to the state Ready To Initialize. The EM federate enters the **Joining** state.
When all participating federates have joined and are in the Ready To Initialize state, the EM federate registers and achieves the "Initialized" synchronization point and transitions to the **Initializing** state. In this state the participating federates are expected to perform their initialization, achieve the synchronization point "Initialized" and transition to the Initialized state.
Once the federation is synchronized the EM federate transitions to the **Suspended** state. Participating federates are expected to remain in the Initialized state and wait for the StartResume or StopFreeze simulation interaction from the EM federate.
The Start and Pause events are the same as in the Start-Stop EM state diagram pattern (see previous paragraph).

On the Stop event the EM federate sends a StopFreeze simulation interaction. It registers and achieves the "Stopped" synchronization point and transitions to the **Stopping** state. The participating federates are expected to stop the time advancement of the simulation, achieve the synchronization point "Stopped" and transition to the Stopped state.
Once the federation is synchronized the EM federate resigns from and destroys the federation execution. The participating federates are expected to resign from the federation execution as well.

In this pattern we have chosen to send both a StopFreeze simulation interaction as well as using a "Stopped" synchronization point in order to provide the possibility to each participating federate to process all interactions and object updates before achieving the synchronization point.

As already mentioned with the previous state control pattern, the Join, Start, Pause and Stop events are typically generated from the graphical user interface of the EM federate. The events Federates Discovered and Federation Synchronized are generated from the RTI.

The EM federate activities and actions are summarized in Figure 11.

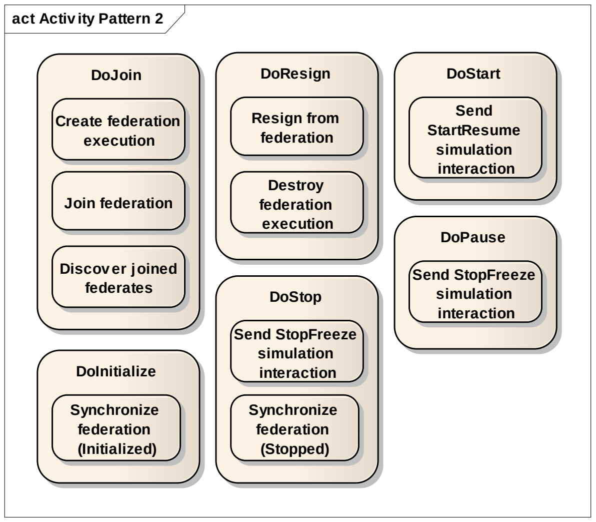

act Activity Pattern 2
DoJoin
Create federation execution
Join federation
Discover joined federates
DoResign
Resign from federation
Destroy federation execution
DoStart
Send StartResume simulation interaction
DoPause
Send StopFreeze simulation interaction
DoStop
Send StopFreeze simulation interaction
Synchronize federation (Stopped)
DoInitialize
Synchronize federation (Initialized)

Figure 11: Start-Stop with initialization activity diagram.

5.3 Start-Stop with Initialization/Reset Pattern

This pattern elaborates further on the Start-Stop with initialization pattern. It adds the possibility to reset the participating federates to their initial state without the need to restart federates. Two additional states have been added to the state control pattern:

- **Stopped**: EM federate is waiting for the Reset or Resign event. Participating federates are in the Stopped state.

- **Resetting**: EM federate is waiting for the participating federates to reset.

The resulting state control diagram is shown in Figure 12.

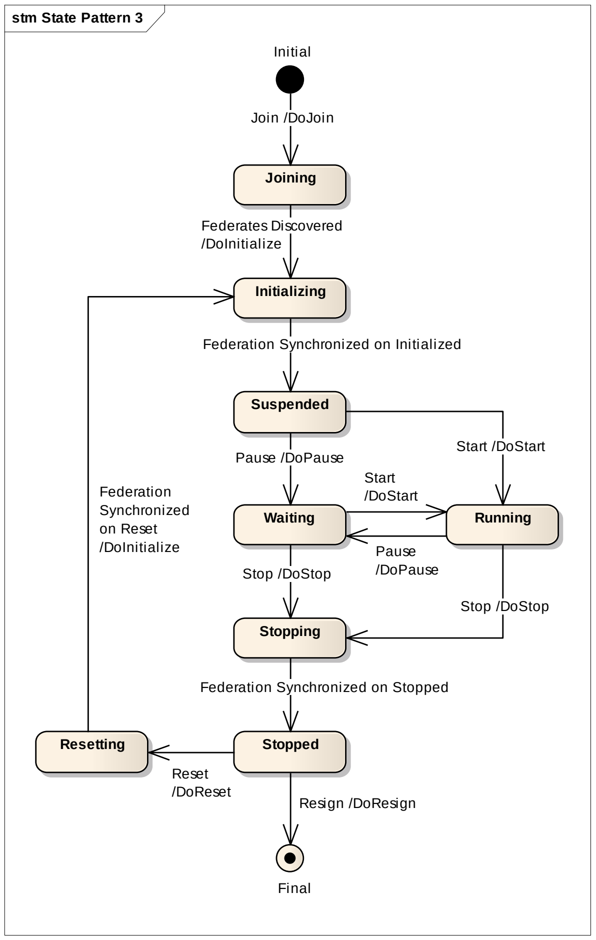


Figure 12: Start-Stop with initialization/reset state diagram.

The EM federate states and activities are the same as in the previous state control pattern, with the addition of the Stopped and Resetting state and the related activities.
On the Reset event the EM federate registers and achieves the "Reset" synchronization point and transitions to the state **Resetting**. In this state the participating federates are expected to perform their reset activity, achieve the synchronization point "Reset" and transition to the state Ready To Initialize.
Once the federation is synchronized the EM federate registers and achieves the "Initialized" synchronization

point and transitions to the **Initializing** state. In this state the participating federates are expected to perform their initialization, achieve the synchronization point "Initialized" and transition to the Initialized state.

The EM federate activities and actions are summarized in Figure 13.

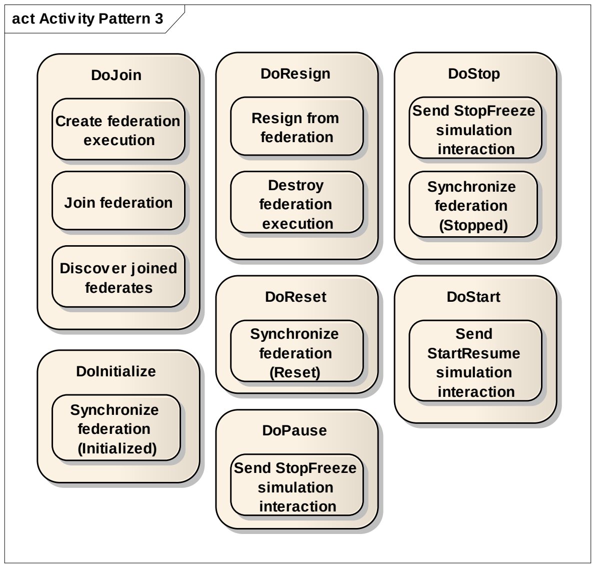

act Activity Pattern 3
DoJoin
Create federation execution
Join federation
Discover joined federates
DoResign
Resign from federation
Destroy federation execution
DoStop
Send StopFreeze simulation interaction
Synchronize federation (Stopped)
DoReset
Synchronize federation (Reset)
DoStart
Send StartResume simulation interaction
DoInitialize
Synchronize federation (Initialized)
DoPause
Send StopFreeze simulation interaction

Figure 13: Start-Stop with initialization/reset activity diagram.

5.4 Iteration Pattern

The state control pattern in Figure 14 shows an iterative state diagram that is typical for a Monte Carlo (stochastic) simulation. This state control pattern is however not supported by the state control pattern for the participating federates shown in chapter 4. The state control pattern for the participating federates needs to be extended for this, or alternatively a new pattern needs to be defined. The EM federate activities and actions are summarized in Figure 15.

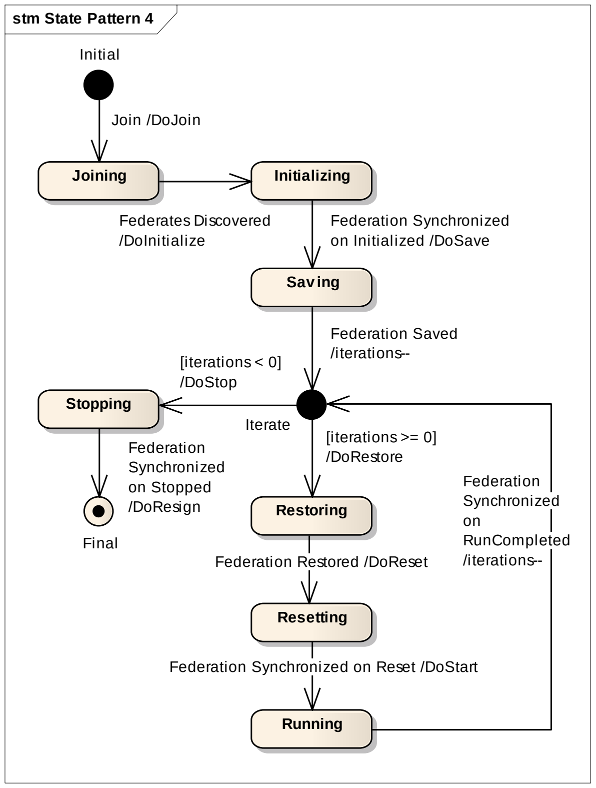
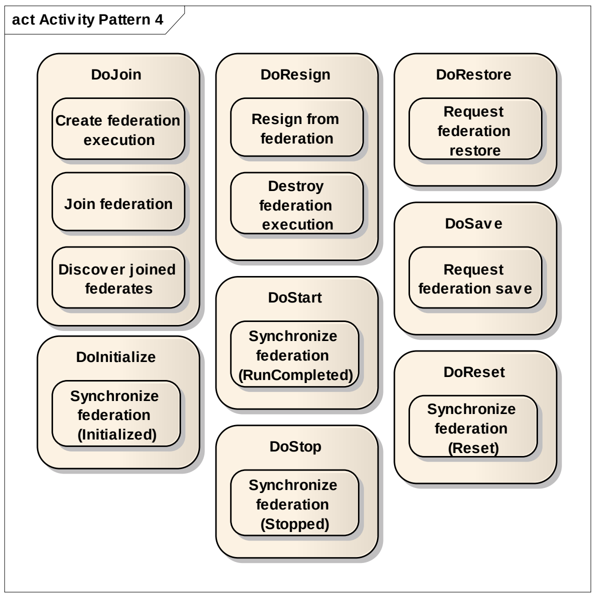

stm State Pattern 4
Initial
Join /DoJoin
Joining
Initializing
Federates Discovered /DoInitialize
Federation Synchronized on Initialized /DoSave
Saving
Federation Saved /iterations--
[iterations < 0] /DoStop
Stopping
Iterate
[iterations >= 0] /DoRestore
Federation Synchronized on Stopped /DoResign
Final
Restoring
Federation Synchronized on RunCompleted /iterations--
Federation Restored /DoReset
Resetting
Federation Synchronized on Reset /DoStart
Running

Figure 14: Iteration state diagram.

Like the previously described Start-Stop patterns there is a **Joining** and **Initializing** state (not described further). Once initialization has completed the EM federate initiates a Federation Save to be able to restore the federation state for each successive iteration later on. It enters the **Saving** state. Participating federates are expected to save their federate state.
Once the federation has been saved the EM federate enters the pseudo state **Iteration**. As long as there are iterations to be done, the EM federate requests a Federation Restore and transitions to the **Restoring** state. Participating federates are expected to restore their federate state.
Once the federation restore has completed the EM federate registers and achieves the "Reset" synchronization point and transitions to the **Resetting** state. In this state participating federates are expected to reset their object instances to the initial state values required for this iteration. A seed and/or iteration number might be involved to bring in perturbations for certain model parameters of the federate (this added complexity has been left out of the state diagram).
Once the reset has finished the EM federate registers and achieves the "RunCompleted" synchronization point and transitions to the **Running** state. Participating federates are expected to start the time advancement of the simulation and achieve the

synchronization point “RunCompleted” once the simulation has completed (for this iteration).
The determination of the achievement of the “RunCompleted” synchronization point may actually be a complex decision making process involving one or more participating federates. The assumption here is that the EM federate is not involved in this decision making process. For example, one of the participating federates could act as an arbiter who determines when the end of a simulation run has been reached (e.g. on missile intercept or timeout). To mark the end of a run the arbiter sends out an interaction to the other participants in order for them to achieve the synchronization point.
When the federation is synchronized the EM federate transitions again to the pseudo state **Iterate**.
When there are no more iterations to go, the EM federate registers and achieves the “Stopped” synchronization point and transitions to the **Stopping** state. In this state participating federates are expected to stop the simulation and achieve the synchronization point.
Once the federation is synchronized the EM federate resigns from and destroys the federation execution, and terminates. The participating federates are expected to resign from the federation execution and terminate as well.


act Activity Pattern 4
DoJoin
Create federation execution
Join federation
Discover joined federates
DoInitialize
Synchronize federation (Initialized)
DoResign
Resign from federation
Destroy federation execution
DoStart
Synchronize federation (RunCompleted)
DoStop
Synchronize federation (Stopped)
DoRestore
Request federation restore
DoSave
Request federation save
DoReset
Synchronize federation (Reset)
Figure 15: Iteration activity diagram.

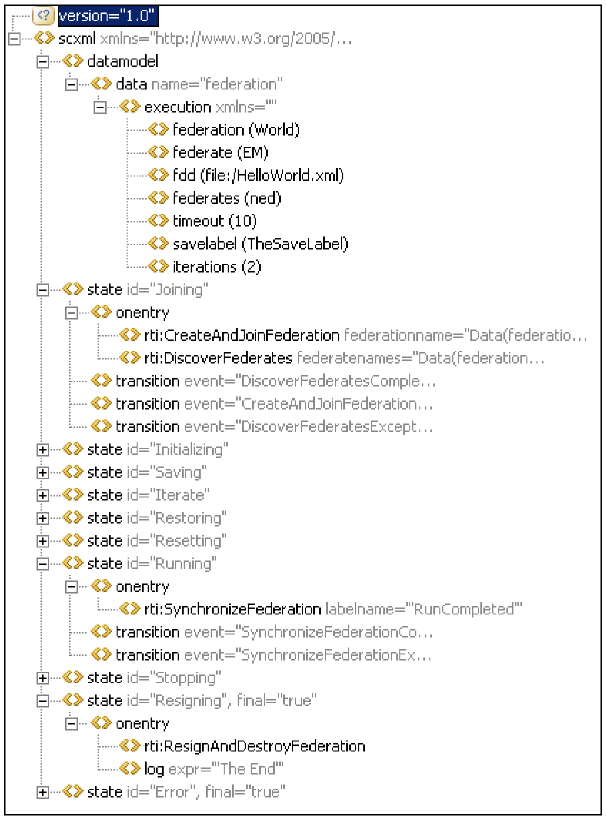
6. State Chart XML

This chapter illustrates with some SCXML snippets how a state control pattern can be translated to an SCXML document with custom actions and events. As an example we use the iteration pattern because this pattern contains the most interesting ingredients w.r.t. the translation to SCXML. The other state control patterns can be mapped to SCXML in a similar way. The next chapter describes an EM federate that is capable of executing this SCXML document.

Note that the space in this paper is too limited to explain all the aspects of SCXML. We restrict ourselves to snippets showing the most relevant elements of SCXML and let the reader refer to reference [5] for more information.

The main elements of the SCXML document (see Figure 16) for the iteration state control pattern are:

- A top level **scxml** element;
- A **data model** element that defines the data objects used in the state machine, such as the names of the federates that will join and the number of iterations to perform;
- A sequence of **state** elements with on-entry actions and with outgoing transitions.


```
version="1.0"
scxml xmlns="http://www.w3.org/2005/...
  datamodel
    data name="federation"
      execution xmlns=""
        federation (World)
        federate (EM)
        fdd (file:/HelloWorld.xml)
        federates (ned)
        timeout (10)
        savelabel (TheSaveLabel)
        iterations (2)
  state id="Joining"
    onentry
      rti:CreateAndJoinFederation federationname="Data(federatio...
      rti:DiscoverFederates federatenames="Data(federation...
    transition event="DiscoverFederatesComple...
    transition event="CreateAndJoinFederation...
    transition event="DiscoverFederatesExcept...
  state id="Initializing"
  state id="Saving"
  state id="Iterate"
  state id="Restoring"
  state id="Resetting"
  state id="Running"
    onentry
      rti:SynchronizeFederation labelname="'RunCompleted'"
    transition event="SynchronizeFederationCo...
    transition event="SynchronizeFederationEx...
  state id="Stopping"
  state id="Resigning", final="true"
    onentry
      rti:ResignAndDestroyFederation
      log expr="'The End'"
  state id="Error", final="true"
```
Figure 16: SCXML document.

The state elements correspond to the states defined in the state control pattern. In the pattern the actions are performed on each transition, but for this SCXML document we have decided to place all actions in the on-entry part of each state element.

The custom actions defined in this SCXML document ultimately map to (Java) code that is part of the SCXML execution environment. This custom code will typically result in an RTI call. The RTI callbacks are also part of the execution environment and result in events for the state machine. This mechanism is explained in more detail in the next chapter. For now it is sufficient to mention that we have the following custom actions and events available in the SCXML document:

Custom action:	**Event:**
`CreateAndJoinFederation`	-
`DiscoverFederates`	`DiscoverFederatesCompleted`
`SynchronizeFederation`	`SynchronizeFederationCompleted`
`SaveFederation`	`SaveFederationCompleted`
`RestoreFederation`	`RestoreFederationCompleted`
`ResignAndDestroyFederation`	-
`ResignFederates`	-
`StartResume`	-
`StopFreeze`	-

The CreateAndJoinFederation action is a combination of CreateFederation and JoinFederation. Similarly the ResignAndDestroyFederation is a combination of ResignFederation and DestroyFederation.

An exception may be raised from a custom action in case of an error. From SCXML point of view an exception is the same as an event, but by giving these events a special name it is possible to handle error cases in the same way as other events.

Now we have a look at some SCXML snippets. We start with the top level element of the SCXML document, followed by the data model and a number of state elements.

6.1 SCXML Element

The top level element of the SCXML document is:

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       xmlns:rti="http://rti.actions/CUSTOM"
       version="1.0"
       initialstate="Joining">
```

The **xmlns** attribute defines the namespace for the custom actions, in this case “rti”. The **initialstate** attribute refers to the initial state of the state machine.

6.2 Data Model Element

The data model defines the data objects that we use in the state machine. It is actually the only place in the SCXML document to memorize state machine data, as custom actions by itself are by definition memory less. The data model is defined as:

```
<datamodel>
  <data name="federation">
    <execution xmlns="">
      <federation>World</federation>
      <federate>EM</federate>
      <fdd>file:/HelloWorld.xml</fdd>
      <federates>ned</federates>
      <timeout>10</timeout>
      <savelabel>TheSaveLabel</savelabel>
      <iterations>2</iterations>
    </execution>
  </data>
</datamodel>
```

The value of a data object can be accessed or updated from the SCXML document via an XPath expression. To access for example the value of the data object “fdd” we use:

```
"Data(federation,'execution/fdd')"
```

6.3 State Elements

The next couple of snippets show the state elements Joining, Initializing, Saving, Iterate and Resigning. The other state elements are similar to these state elements and the reader can easily infer them.

The initial state of the state machine is the **Joining** state. The Joining state has two on-entry actions, namely: *CreateAndJoinFederation* and *DiscoverFederates*. For the second action the event *DiscoverFederatesCompleted* is raised when all listed federates have joined. This event causes a transition to the next state, Initializing. An exception causes a transition to the Error state.

```
<state id="Joining">
  <onentry>
    <rti:CreateAndJoinFederation
      federationname=
       "Data(federation,'execution/federation')"
      Federatename=
        "Data(federation,'execution/federate')"
      fdd=
        "Data(federation,'execution/fdd')" />

    <rti:DiscoverFederates
      Federatenames=
        "Data(federation, 'execution/federates')"
      timeout=
        "Data(federation, 'execution/timeout')" />
  </onentry>

  <transition
    event="DiscoverFederatesCompleted"
    target="Initializing"/>
  <transition
    event="CreateAndJoinFederationException"
    target="Error"/>
  <transition
    event="DiscoverFederatesException"
    target="Error"/>
</state>
```

The **Initializing** state has one on-entry action called *SynchronizeFederation*. This action registers a synchronization point. Once the federation is synchronized, that is the participating federates have achieved this synchronization point, the event

SynchronizeFederationCompleted is raised, causing a transition to the Saving state.

```
<state id="Initializing">
  <onentry>
    <rti:SynchronizeFederation
      labelname="'Initialized'"/>
  </onentry>

  <transition
    event="SynchronizeFederationCompleted"
    target="Saving"/>
  <transition
    event="SynchronizeFederationException"
    target="Error"/>
</state>
```

The **Saving** state has one on-entry action called *SaveFederation*, to initiate a Federation Save. The event *SaveFederationCompleted* triggers a transition to the next state, Iterate.

```
<state id="Saving">
  <onentry>
    <rti:SaveFederation
      savelabel=
        "Data(federation, 'execution/savelabel')" />
  </onentry>

  <transition
    event="SaveFederationCompleted"
    target="Iterate"/>
  <transition
    event="SaveFederationException"
    target="Error"/>
</state>
```

The **Iterate** state does a count down on the number of iterations. In the on-entry part the number of iterations is decremented by one. Depending on the current value a transition is made to either the Restoring state or the Stopping state.

```
<state id="Iterate">
  <onentry>
    <assign
      location=
        "Data(federation, 'execution/iterations')"
      Expr=
        "Data(federation, 'execution/iterations')-1"
/>
  </onentry>

  <transition
    cond="Data(federation, 'execution/iterations') >=
0"
    target="Restoring" />
  <transition target="Stopping" />
</state>
```

The **Resigning** state is a final state. It has an on-entry action *ResignAndDestroyFederation* to resign from and destroy the federation execution, as well as logging a final message.

```
<state id="Resigning" final="true">
  <onentry>
    <rti:ResignAndDestroyFederation />
    <log expr="'The End'"/>
```

```
  </onentry>
</state>
```

7. Execution Manager Federate

The snippits as described in the previous chapter are part of an SCXML document that specifies a state diagram. TNO has developed an EM federate that can execute such an SCXML document by using the Apache Commons SCXML component [7].

Apache Commons SCXML is an implementation of the W3C SCXML specification aimed at creating and maintaining a Java SCXML engine. It is capable of executing a state machine defined using an SCXML document, and abstracts out the environment interfaces. Commons SCXML is a reusable Java component from the Apache Commons project, which is an open-source project of the Apache Software Foundation.

The TNO EM federate consists of the five packages as shown in Figure 17.

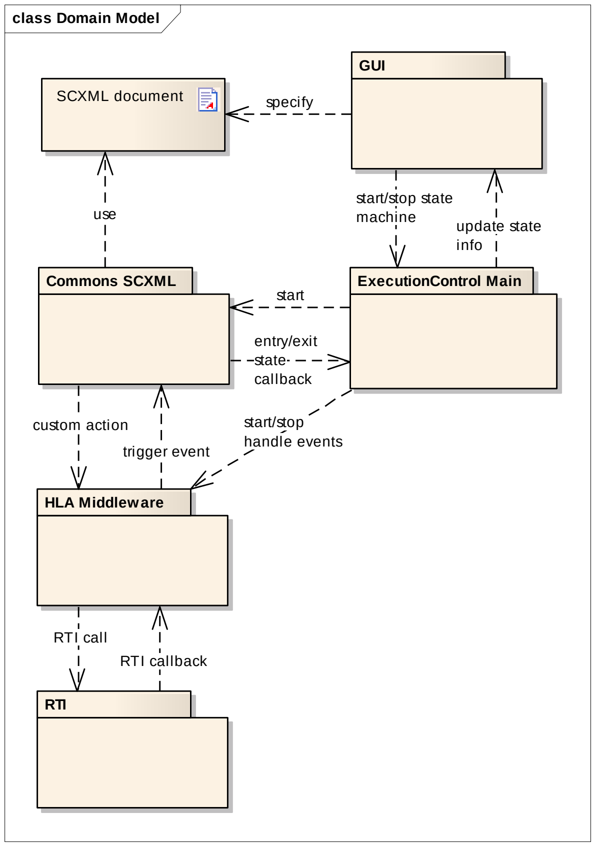


Figure 17: Package diagram of the TNO EM federate.

The Commons SCXML and the RTI packages are third party libraries, while the GUI, ExecutionControl and

HLA Middleware packages are developed for this use case. The *Model-View-Controller* design pattern [8] has been applied by dividing the federate into a:

1. Model:

The core functionality consists of the Commons SCXML, the HLA Middleware, and the RTI library. Commons SCXML executes the state machine by performing custom actions as described in the SCXML document until it has to wait on an event to be triggered by the HLA Middleware. The triggering of the events keeps the state machine running, because it will cause state transitions and new actions to be performed. The HLA Middleware provides an abstraction of the RTI programming interface that can be used easily by the custom actions. A custom action can correspond to one or more RTI calls. Besides hiding the interface details of the RTI, the HLA Middleware implements an event handling loop that receives and processes the RTI callbacks until a final state has been reached or a stop command has been received from the user interface. An RTI callback can cause the triggering of an event. Commons SCXML then decides whether this event causes a state transition.

2. View:

The GUI lets the user select the SCXML document, provides buttons to start and stop the state machine, and displays the current state name.

3. Controller:

The ExecutionControl initializes Common SCXML with the SCXML document and the list of custom actions, handles the user input from the GUI, starts the state machine by starting the SCXML executor, starts the event handling loop of the HLA Middleware, stops the state machine (when the user wants to stop it before a final state has been reached) by stopping the event handling loop of the HLA Middleware, handles state change notifications from Commons SCXML and sends the new state info to the GUI.

The action-event loop is the core of the federate executing a state diagram: custom actions performed by Commons SCXML result in RTI calls, while RTI callbacks processed by the HLA Middleware result in the triggering of events, etc. Also RTI exceptions raised during RTI calls can cause the triggering of exception events.

For the SynchronizeFederation action this loop is illustrated in more detail in Figure 18. The action results in multiple RTI calls and callbacks and it ends with a single SynchronizeFederationCompleted event. If something would have gone wrong, then it should end with a SynchronizeFederationException event (this is not shown in the figure).

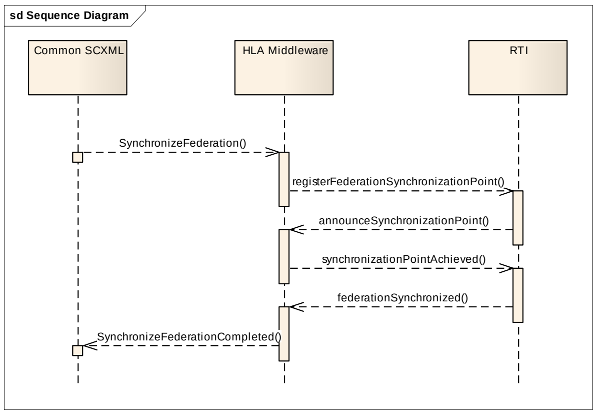

sd Sequence Diagram
Common SCXML
HLA Middleware
RTI
SynchronizeFederation()
registerFederationSynchronizationPoint()
announceSynchronizationPoint()
synchronizationPointAchieved()
federationSynchronized()
SynchronizeFederationCompleted()

Figure 18: Sequence diagram of the SynchronizeFederation action and resulting event.

The SCXML executor has no method to stop the execution of the state machine before a final state has been reached. However, the same result can be reached by stopping the event handling loop of the HLA Middleware, such that no events will be triggered anymore and the state machine becomes inactive. If the federate is still joined at that moment the HLA Middleware will resign from the federation and the state machine is able to be restarted without any problem.

Commons SCXML allows an application to change the state machine at run-time. States or transitions for example may be added to the state machine through the Commons SCXML programming interface. From the Commons SCXML programming interface it is also possible to manipulate the data model, thus allowing data object values to be changed at run-time (by for example a custom action). All in all, Commons SCXML provides a rich programming interface to manipulate the state machine and data model at run time.

8. Conclusions and Further Developments

The main conclusions are:

- Three of the four execution management state control patterns shown in this paper correlate very well with the proposed participating federate state control pattern. For the iteration pattern the participating state pattern needs to be extended, or alternatively a new pattern needs to be created.
- As the number of different execution state control patterns and variants of the same pattern increase it is important to have a taxonomy. This is currently lacking.

- SCXML provides a formalism for state charts and is suitable as specification language for HLA execution state control patterns and as exchange format between different execution manager federate applications. However, the naming and meaning of custom actions and events need to be standardized. This is required to enable the exchange of SCXML documents across different SCXML execution environments.
- SCXML is human readable. No graphical editor is required to create an SCXML document. The snippets shown in this paper were all created in WordPad.
- SCXML state names are global and must be unique. Also nested states must have unique names. It is possible to include state charts from other SCXML documents. For example, smaller patterns may be defined in separate documents and included from the main document. The naming restriction however limits this kind of reuse of state charts.
- SCXML matures to a W3C standard. Already execution environments are available for SCXML. For Java there is the open source project Apache Commons SCXML (see [7]). And for C++ there is a commercial product from Sidema (see [9]).

Future work includes:

- The patterns presented in this paper are defined from both the EM federate point of view and participating federate point of view. This paper gives first proposals for patterns from both views. It is clear that different patterns are needed depending on the type of application. For example, patterns that model late joining and early leaving federates, patterns for the Simulation Management family of interactions/PDUs and patterns for tightly or loosely coupled federations. Additional patterns may involve more complex modeling using concurrent states.
- Suitability of the proposed design patterns needs to be investigated by corresponding experiments. First experiments are being performed under MSG-068.
- The BOM concept [10] formalizes the way a conceptual model is described and part of this definition are the pattern of interplay and the state machine template components. We believe that the state control patterns described in this paper complement these BOM concepts. It would be a step forward if the state machine defined in a BOM can be translated automatically into an (executable) SCXML document, thus making a shift from the reuse of federate code to the reuse of simulation models by using model driven development techniques [11].

9. References

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language: “Towns, Buildings, Construction”. Oxford University Press, New York, 1977.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: “Design patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, ISBN 0-201-63361-2, 1995.

[3] David Harel: “Statecharts: A visual formalism for complex systems”. Science of Computer Programming, Volume 8 , Issue 3, ISSN 0167-6423, June 1987.

[4] David Harel and Michal Politi: “Modeling Reactive Systems with Statecharts”. McGraw-Hill, ISBN 0-07-026205-5, 1998.

[5] W3C SCXML: http://www.w3.org/TR/scxml.

[6] REC-XML-19980210, Extensible Markup Language (XML) 1.0, W3C Recommendation, Feb. 1998.

[7] Apache Commons SCXML project: http://commons.apache.org/scxml.

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: “Pattern-Oriented Software Architecture – A System of Patterns”. Wiley, ISBN 0-471-95869-7, 1996.

[9] Sidema: http://www.sscxml.com.

[10] SISO-STD-003-2006, SISO Base Object Model (BOM) Template Specification, March 2006.

[11] Roger Jansen, Louwrens Prins, Wim Huiskamp, “Template Driven Code Generator for HLA Middleware”, Simulation Interoperability Workshop, Spring 2007 (07F-SIW-038)

[12] MSG-052 Portal: http://msg052.smart-lab.se.

[13] MSG-068 Portal: subspace under MSG-052 Portal.

Author Biographies

TOM VAN DEN BERG is scientist in the M&S department at TNO Defence, Security and Safety, The Netherlands. He holds an M.Sc. degree in Mathematics and Computing Science from Delft Technical University. His research area includes distributed processing and simulation systems, software architectures and software process improvement.

ROGER JANSEN is a member of the scientific staff in the M&S department at TNO Defence, Security and Safety in the Netherlands. He holds an M.Sc. degree in Computing Science and a Master of Technological Design (MTD) degree in Software Technology, both from Eindhoven University of Technology, The Netherlands. He works in the field of distributed simulation and his research interests include distributed computing and simulation interoperability.

HARTMUT UFER studied physics at the University of Osnabrück, Germany, and achieved his master degree in 1992. During his work as a scientist at the university he specialized in theoretical solid state physics. Since 1998, Mr. Ufer works in the field of distributed simulation, and, since 2002, works as a senior software engineer at IABG mbH, Germany.