DEPARTMENT OF DEFENCE
DEFENCE SCIENCE & TECHNOLOGY ORGANISATION
DSTO

DSTO

Execution Management in the Virtual Ship Architecture Issue 1.00

Anthony Cramp and John P. Best

DSTO-GD-0258

Execution Management in the Virtual Ship Architecture Issue 1.00

Anthony Cramp and John P. Best

Maritime Operations Division
Aeronautical and Maritime Research Laboratory

DSTO-GD-0258

ABSTRACT

The Virtual Ship is an application of the High Level Architecture (HLA) in which simulation models that represent the components of a warship are brought together in a distributed manner to create a virtual representation of a warship. It is necessary to impose a degree of control over the flow of execution of these distributed components, beyond that provided by the HLA run-time infrastructure (RTI). This control is referred to as execution management and described in this document.

RELEASE LIMITATION

Approved for public release

DEPARTMENT OF DEFENCE
DEFENCE SCIENCE & TECHNOLOGY ORGANISATION
DSTO

Published by

DSTO Aeronautical and Maritime Research Laboratory
PO Box 4331
Melbourne Victoria 3001 Australia

Telephone: (03) 9626 7000
Fax: (03) 9626 7999

AR-011-613
October 2000

APPROVED FOR PUBLIC RELEASE

Execution Management in the Virtual Ship Architecture Issue 1.00

Executive Summary

The Virtual Ship is an application of the High Level Architecture (HLA) in which simulation models that represent the components of a warship are brought together in a distributed manner to create a virtual representation of a warship.

A requirement for any distributed system is coordination of the activities of the components. In the context of a HLA federation, coordination of the federates is required, and the RTI provides a basic set of services to facilitate this. For example, the federation management services provide the means to create and destroy federations and by which individual federates join and resign from a federation execution. The time management services provide for coordination of the time advances of federates.

However, there is a requirement to impose a level of coordination above that provided by the RTI services. It is required to

1. Provide an assurance that all required simulation models have joined the federation execution before the simulation proceeds,
2. Provide an assurance that a specified initial state is established before the simulation proceeds,
3. Provide a mechanism whereby multiple replications of a scenario can take place within a single federation execution in order to support Monte Carlo techniques,
4. Provide to the federation a random number seed, if required,
5. Provide to the federation a unique identifier for each simulation of a scenario.

It is these services that are not provided by the RTI but are considered necessary to enable a Virtual Ship federation execution to occur in a controlled way and a repeatable way, if desired. It is the provision of these services that we refer to as execution management.

This document describes the Virtual Ship concept for execution management in detail. The concept essentially defines the high level flow of a Virtual Ship federation execution and this is described in terms of the federation undergoing transitions amongst a finite collection of states. To implement the concept requires three components: a federate known as the Virtual Ship Execution Manager (VSEM), a file input to the VSEM that details how the federation execution should take place and the functionality built into the other federates to interact with the VSEM.

Page deliberately left blank

Revision Record

The controlled version of this document resides in the Virtual Ship Project System Analysis and Design Folder. Any copy of this document, electronic or otherwise, should be considered uncontrolled. Amendment shall be by whole document replacement. Proposals for change should be forwarded to the issuing authority.

Issue	Date	Author	Description of Revision
1.00	20Jul00	A. Cramp, J. Best	First issue.

Page deliberately left blank

Contents

Page deliberately left blank

1 Introduction

A requirement for any distributed system is coordination of the activities of the components. In the context of a HLA federation, coordination of the federates is required, and the RTI provides a basic set of services to facilitate this. For example, the federation management services provide the means to create and destroy federations and by which individual federates join and resign from a federation execution. The time management services provide for coordination of the time advances of federates.

However, it is evident that there is a requirement to impose a level of coordination above that provided by the RTI services.

In the first instance, although federates may require data from other federates, they are typically designed to continue to operate in the absence of such data and there is no RTI service that provides advice that required data sources are absent. The essential problem is one of ensuring that all federates required for a federation execution have joined before the simulation proceeds.

It is also typically required to have a mechanism whereby a specified initial state for the federation can be defined and an assurance obtained that the specified initial state is established before the simulation proceeds.

These two control features are essential in the event that fully repeatable federation executions are to take place. There are two requirements for repeatability; sameness of initial conditions and sameness of inputs throughout the simulation. The requirements outlined in the previous paragraph contribute to ensuring that the initial conditions of the federation are identical and replicable. The use of the time management services provides the other component of repeatability under conditions of identical inputs throughout the federation execution.

It should be noted that the Virtual Ship may be used to support real time simulation in which time management services are not used. Even under such circumstances the conditions detailed previously are required to ensure that the federates interact in a meaningful fashion.

The requirement for the Virtual Ship Architecture [1] calls for scenario replication in order to support Monte Carlo techniques. Because of the intrinsic distributed nature of a Virtual Ship and the difficulty and time required to remotely activate federates, the solution has been to provide for replication of scenario execution within a single federation execution. This involves deleting object instances at the conclusion of a scenario execution followed by recreation of the scenario initial conditions. It is particularly critical to reset the federation time to zero.

Support for multiple scenario executions within a single federation execution requires that a mechanism be put in place whereby each replication can be uniquely identified. This facilitates the identification of data sets, and other simulation outputs, with a particular scenario.

The use of Monte Carlo techniques also typically involves the use of random variations of conditions from one scenario to another. Hence it is required to generate and promulgate random number seeds throughout the federation.

It is these services that are not provided by the RTI but are considered necessary to enable a Virtual Ship federation execution to occur in a controlled way and a repeatable way, if desired. It is the provision of these services that we refer to as execution management.

This document describes the Virtual Ship concept for execution management in detail. The concept essentially defines the high level flow of a Virtual Ship federation execution and this is described in terms of the federation undergoing transitions amongst a finite collection of states. To implement the concept requires three components: a federate known as the Virtual Ship Execution Manager (VSEM), a file input to the VSEM that details how the federation execution should take place and the functionality built into the other federates to interact with the VSEM.

The flow of a managed federation is shown in Figure 1 and described as follows. The initial phase consists of all federates joining the federation. The VSEM prevents the federation proceeding until all federates have joined. The next phase is an initialisation phase, during which time federates perform internal initialisation, such as establishing their time management policies. The next phase is a federation save. Each of the federates saves its initial state, which should be null in the sense that no entities exist. In addition, the initial time of the federation is saved. It is the use of the save functionality that enables multiple replications of a scenario within a single federation execution.

Following this phase is a restore phase, where the initial state of the federation is restored and the federation time set to its initial value. Having restored the federation the VSEM may provide to the federates a descriptor and random number seed for this replication of the scenario. The federation now enters the simulate phase and all federates behave according to the entities they are representing. During this phase, the VSEM may send messages to the federates advising them to create composite and component entities. These entities may be created at any time during the scenario, but those created at initial time constitute the initial conditions of the federation.

It is this process that provides an assurance that the entities required within a scenario have been created. These messages are sent in time stamp order (TSO) and all federates that are time constrained will receive them in accordance with the time management algorithms. If the federates that receive these messages are time regulating they will respond to entity creation messages by registering object instances and providing an initial attribute update, which is sent TSO. Hence, a federation in which all federates are time regulating and time constrained cannot proceed until all requested entities have been created and their attributes updated.

At the conclusion of the scenario, which is determined on the basis of elapsed time, a scenario shutdown phase occurs, when each federate may write scenario specific data to file and perform any other task required for a graceful shutdown of a scenario.

If the scenario is to be repeated the restore phase in executed next, and the process repeats as many times as requested. When the final iteration of the scenario is complete, a federation shutdown phase occurs, during which all federates resign from the federation and it is destroyed.

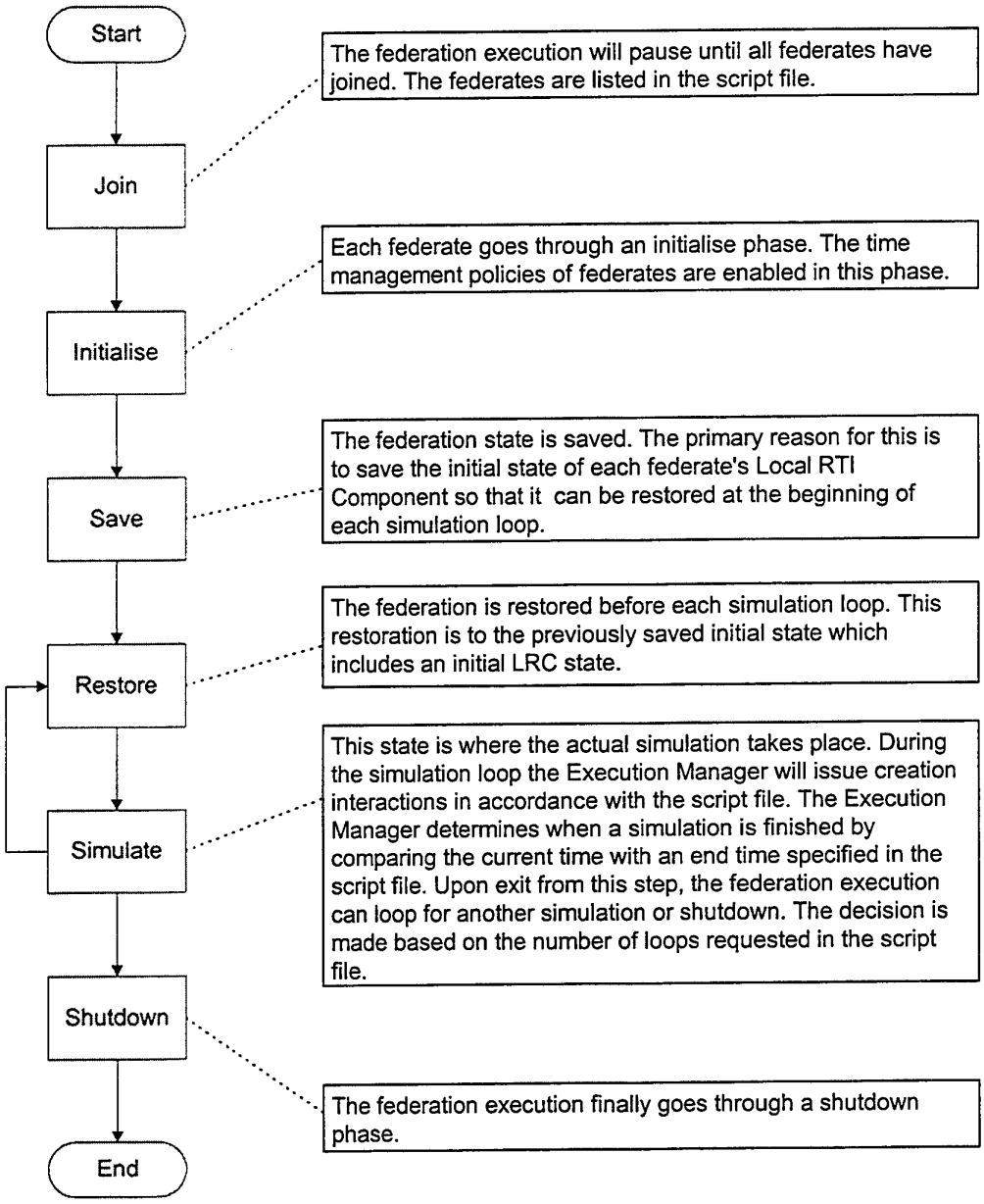

Figure 1: Flow of an execution managed federation execution

The VSEM script file provides the specific details of this process. In particular, it lists the federates that make up the federation, the number of loops of the scenario that are to be executed, whether scenario descriptions and random number seeds are required and the list of composite and component entities that are to be created during the scenario, along with the time at which they are created. The scenario end time is also specified.

As detailed above, the federates receive and respond to messages from the VSEM. The VSEM additionally makes use of synchronisation points to control the flow of execution through the various phases. Hence, federates that interact with the VSEM are required to be constructed to respond to the interactions sent by it, and to respond appropriately to the synchronisation points.

The remainder of this document describes the concept for execution management in detail.

2 Virtual Ship Execution Manager (VSEM)

The VSEM is the controlling entity of a Virtual Ship federation execution. It initialises transitions between the states described above and monitors the federation for the completion of each phase. The VSEM is responsible for interpreting the script file that is used to describe the federation execution. Based on the contents of this script file, the VSEM will supply the federation with a scenario description and random number seed for each pass through a scenario iteration (if specified) and will request the creation of entities at the appropriate times.

The VSEM achieves its control of a federation execution through the use of services specified in the HLA Interface Specification [Draft IEEE 1516.2]. This implies that the VSEM is a federate and must exist in the managed federation execution.

Federation management services are used to provide state transition control. Synchronisation points, saves and restores are registered by the VSEM to initiate a state transition. Notification from the RTI that each of these control mechanisms has completed allows the VSEM to identify when a state has completed. This notification will only be generated once each federate participating in the federation execution has acknowledged state completion. Acknowledgment is performed through a call to the appropriate RTI ambassador service.

The only control point that does not use the federation management services is completion of the Join state. Identifying completion of this state is performed by the VSEM discovering instances of the Manager.Federate object class. The Manager.Federate object class forms a part of the Management Object Model (MOM). Instances of this object class are registered by the RTI for each federate that is joined to the federation execution. On discovery, the VSEM requests an update of the FederateType attribute. The value of this attribute is the name a federate used when it joined the federation execution. By comparing these names with a list of federates that are to exist in the federation, specified in the script file, the VSEM can determine when all the federates have joined.

In order for the VSEM to discover instances of the Manager.Federate object class it must subscribe to this object class. This gives the object class structure table for the VSEM's Simulation Object Model (SOM) shown in Table 1.

Level 1	**Level 2**
Manager (N)	Federate (S)

Table 1: VSEM SOM Object Class Table

During a scenario iteration, the VSEM will notify participating federates of the scenario description and random number seed (if specified in the script file) and will request federates to create entities. Participating federates are federates that are able to respond to execution management messages and are described in section 3. These messages are communicated using interactions. The interaction class structure table of the VSEM's SOM is shown in Table 2.

Level 1	**Level 2**
ExecutionManager (N)	CreateCompositeEntity (I)

	CreateComponentEntity (I)
	SetScenarioDescription (I)
	SetRandomNumberSeed (I)
	ExecutionManagementError (R)

Table 2: VSEM SOM Interaction Class Table

The interactions are grouped under the common super class ExecutionManager. Interactions below this are:

- *CreateCompositeEntity, CreateComponentEntity:* Requests the receiving federate to create a composite or component entity with the information provided in the parameters of the interaction. These interactions can be scripted in a script file and could possibly be initiated through user input.
- *SetScenarioDescription:* The VSEM uses this interaction to notify participating federates of a string that can be used to identify a particular scenario within a single federation execution.
- *SetRandomNumberSeed:* Used to notify participating federates of a number that can be used to seed a random number generator.

There also exists an interaction that can be used by remote federates to inform the VSEM that the remote federate cannot comply with a VSEM request:

- *ExecutionManagementError:* Used by a participating federate to notify the VSEM that an error has occurred in the execution management process. (Not currently implemented).

The parameters for each of these interactions are presented in Table 3.

Class	Parameter	Datatype
CreateCompositeEntity	ClassName	string
	ObjectInstanceName	string
	Federate	string
	ConfigurationFile	string
	AttributeNames	string
	AttributeValues	string
CreateComponentEntity	ClassName	string
	ObjectInstanceName	string
	Federate	string
	ConfigurationFile	string
	ParentObjectInstanceName	string
	AttributeNames	string
	AttributeValues	string
SetScenarioDescription	Description	string
SetRandomNumberSeed	Seed	unsigned long
ExecutionManagementError	Error	string
	Information	string

Table 3: VSEM SOM Parameter Table

The parameters of the CreateCompositeEntity interaction are described as follows:

- *ClassName:* Defines the subclass of the CompositeEntity object class that is to be created.
- *ObjectInstanceName:* Specifies the name that should be used by the receiving federate to register the created object instance.
- *Federate:* The name of the federate responsible for creating the object instance.
- *ConfigurationFile:* The name of a file that can be used to initialise the created object instance. This parameter is a string that may or may not be recognised by the receiving federate. If the receiving federate does make use of a configuration file and a file of the specified name cannot be found, an execution management error is deemed to have occurred.
- *AttributeNames, AttributeValues:* Specify initial values for each of the attributes that appear in the FOM for the object class being created. These parameters are sent as strings with a '$' character delimiting individual values. For example, suppose values for attributes Name and Position are sent with values "Anzac" and 6378137.0, 0.0, 0.0. The AttributeNames parameter is $Name$Position$ and the AttributeValues parameter is $Anzac$6378137.0$0.0$0.0$. This data format will be changed to conform to the Virtual Ship rule regarding the communication of complex data types.

The parameters for the CreateComponentEntity interaction are the same as for CreateCompositeEntity but applied to the ComponentEntity object class branch of the VS-FOM. There is an additional attribute:

- *ParentObjectInstanceName:* This is the object instance name of the composite entity that acts as the parent for the component entity being created. This is the same name as the ObjectInstanceName parameter specified for the CreateCompositeEntity interaction that is to create this components parent.

The sole parameter for the SetScenarioDescription interaction is a string that may be used to identify a particular scenario. The VSEM provides no assurance that this string is unique. It is the responsibility of those drafting the script file to ensure that this string is unique, if required.

The only parameter for SetRandomNumberSeed is Seed, which defines a value that can be used to initialise random number generators.

The parameters for the ExecutionManagementError interaction are (not implemented):

- *Error:* A string describing the error that occurred.
- *Information:* A string providing additional information concerning the error that occurred.

The process of the VSEM is shown in Figure 2, Figure 3 and Figure 4. Each figure describes a specific area of the VSEM process at progressively finer detail. Figure 2 presents the process of the VSEM at a high level. Figure 3 details the process used by the VSEM for looping multiple times in a single federation execution. Figure 4 details the process of a single scenario within a federation execution.

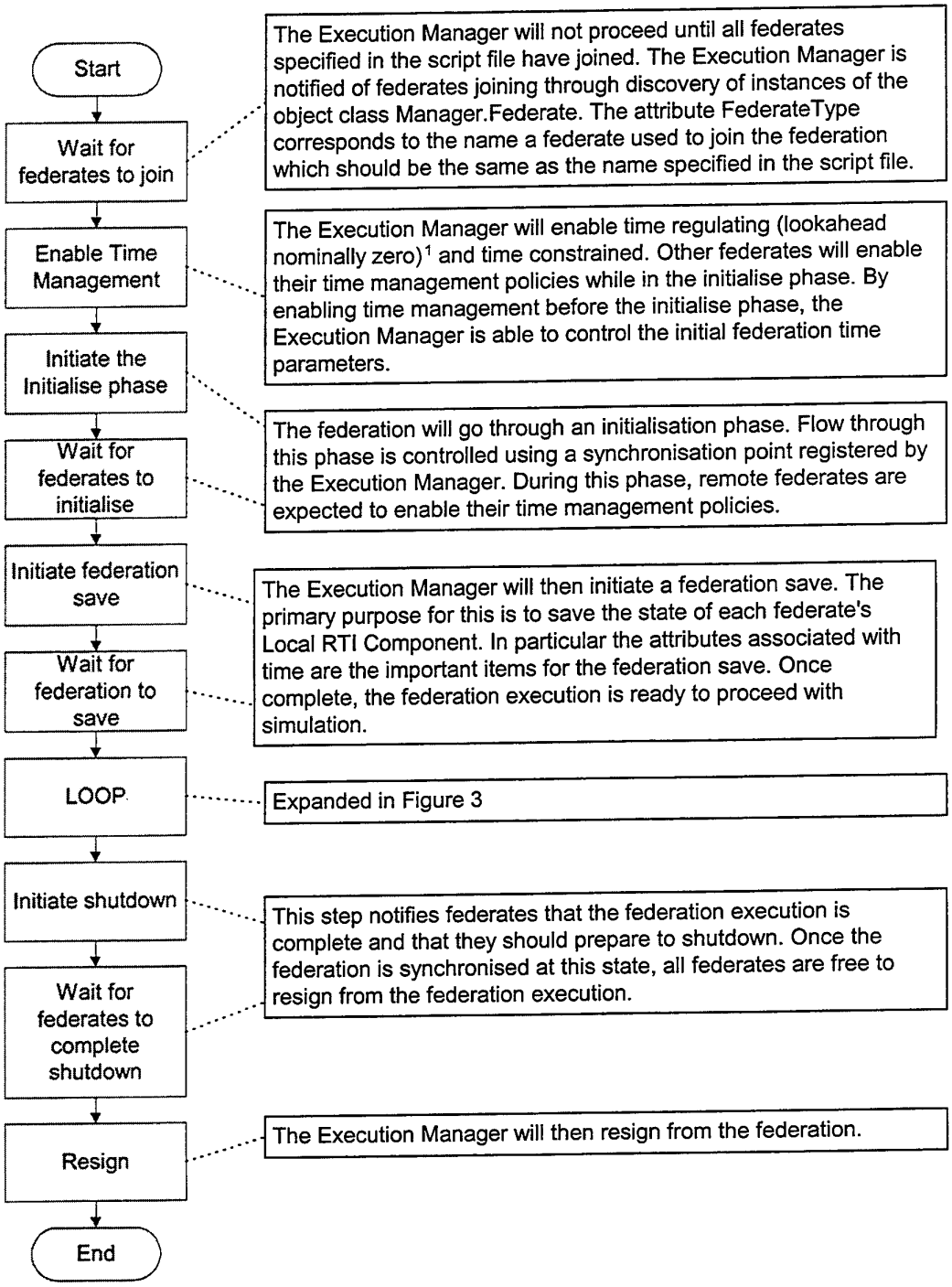

Figure 2: Execution Manager Flow : Overview

1. The DMSO RTI 1.3v6 adds a value of 10^{-9} to a requested lookahead. Thus requesting a lookahead of 0 is actually requesting a lookahead of 10^{-9}.

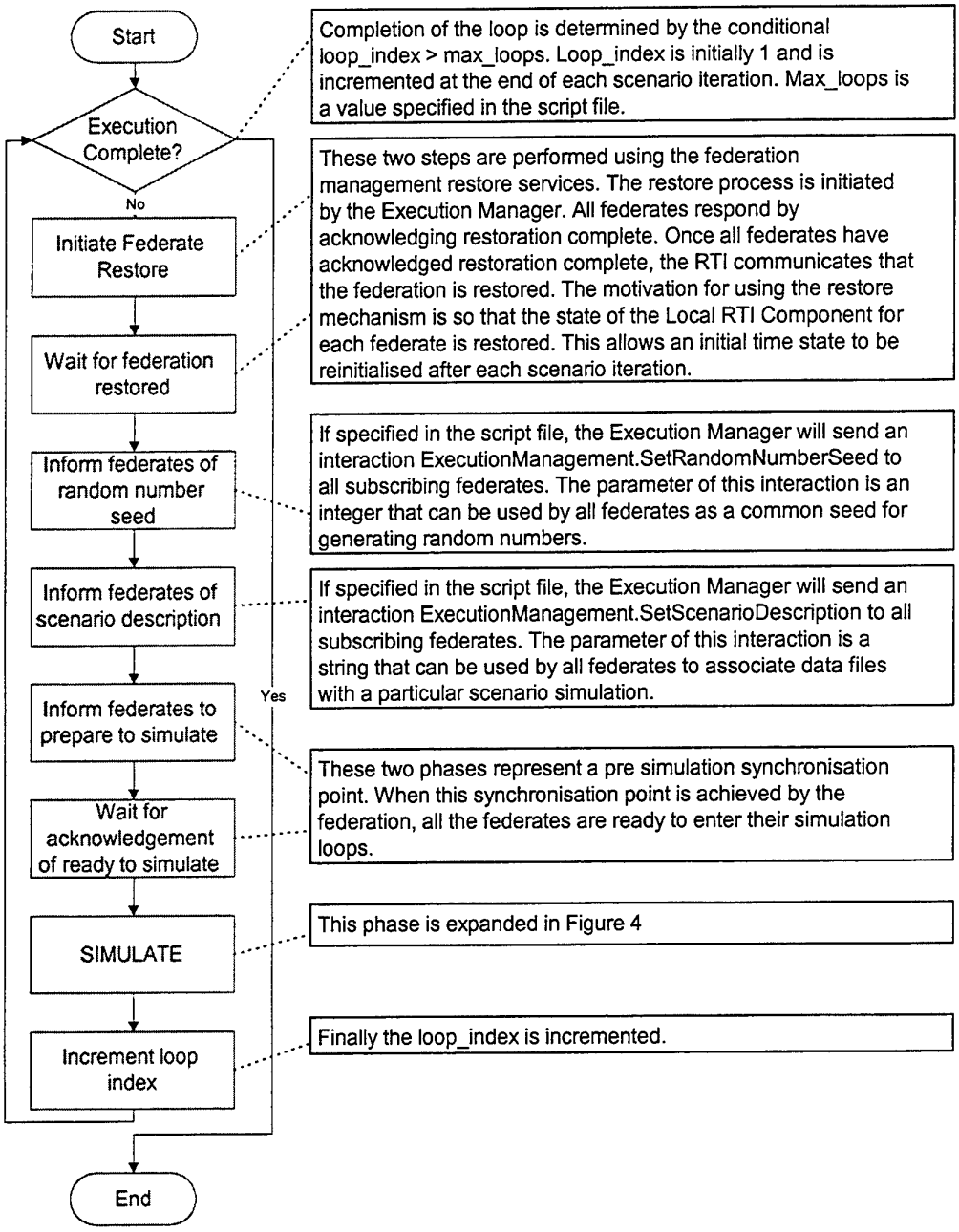

Figure 3: Execution Manager Flow : Loop

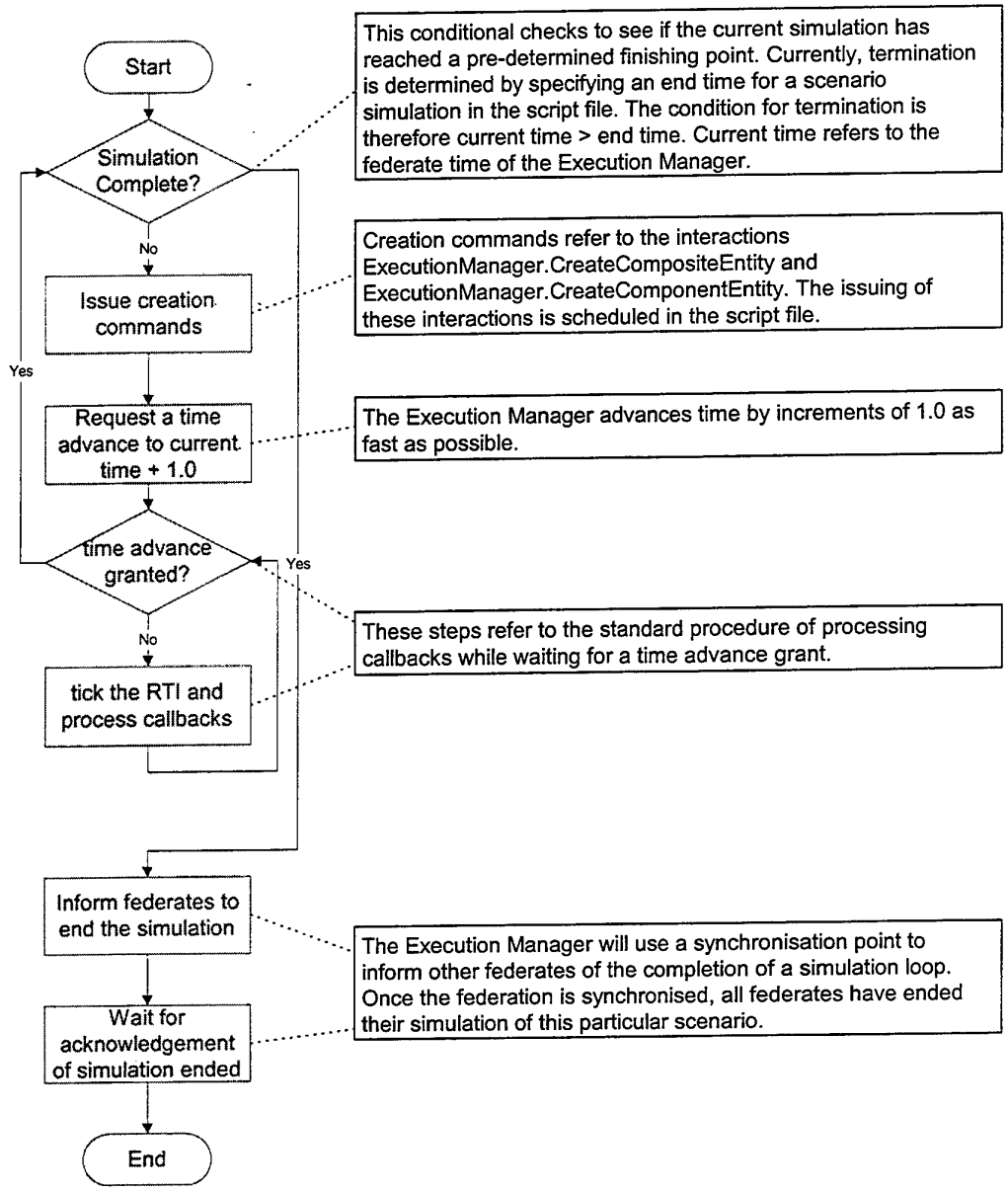

Figure 4: Execution Manager Flow : Simulate

3 Other Federates

Other federates refer to all federates that are members of a Virtual Ship federation, apart from the VSEM. There are two types of other federates in the Virtual Ship implementation of Execution Management, participating and auxiliary federates.

3.1 Participating federates

Participating federates are federates that subject themselves to the controls of the Virtual Ship Execution Manager, and hence are also referred to as execution managed federates. In order for these federates to be able to respond to the controls of the VSEM, they must be able to handle the registration of synchronisation points, federation saves and federation restores.

The VSEM uses interactions to inform participating federates of scenario descriptions and random number seeds and to request participating federates to create entities. In order for a participating federate to receive this information, they must subscribe to these interactions (specified in Table 2) and this is reflected in the participating federate's SOM, shown in Table 4.

Level 1	Level 2
ExecutionManager	CreateCompositeEntity (R)
	CreateComponentEntity (R)
	SetScenarioDescription (R)
	SetRandomNumberSeed (R)
	ExecutionManagementError (I)

Table 4: Extract from Interaction Class Structure Table for a Participating Federate's SOM

Only those interactions that the participating federate can react to need be in the SOM. For example, if the federate does not create composite entities, there is no need for the ExecutionManagement.CreateCompositeEntity interaction to appear in the SOM.

Participating federates will generally be able to create and model entities so at least one of ExecutionManagement.CreateCompositeEntity or ExecutionManagement.CreateComponentEntity will appear in the federate's SOM. As there may be many federates capable of receiving these interactions, a federate will need to determine whether or not the interaction is meant for it. This can be done by comparing the Federate parameter of the interactions with the receiving federate's name. To facilitate this checking, the VSEM encodes the name of the intended federate in the tag of the `sendInteraction` RTI service and is communicated to the receiving federate in the tag of the `receiveInteraction` federate ambassador callback, negating the need to process the parameter handle value pair set.

The current implementation of the VSEM uses an end time to indicate when a particular scenario is complete. Once the VSEM reaches this end time it will begin the transition to ending the simulation state. For participating federates to see this end time at the correct time they need to execute in lockstep with the VSEM. This is achieved by the federates using time regulating and time constrained as their time management policies.

Although the current implementation of the execution management concept is tailored for federates that are time regulating and time constrained, the concept is seen to be sufficiently general to allow for the construction of execution managed federations consisting of federates using arbitrary time management policies. The difference between such an implementation and the current implementation will be the means used for determining when to terminate a scenario iteration. Several alternatives to the current time based approach are given in section 5.2.

Participating federates in an execution managed federation execution are specified in the script file using the keyword "ParticipatingFederate". See section 4 for more information on the syntax of the script file.

The flow of an execution managed federate is shown in Figure 5 and Figure 6. Figure 5 provides a high level description of the process of a participating federate and Figure 6 elaborates on the simulate phase of a participating federate.

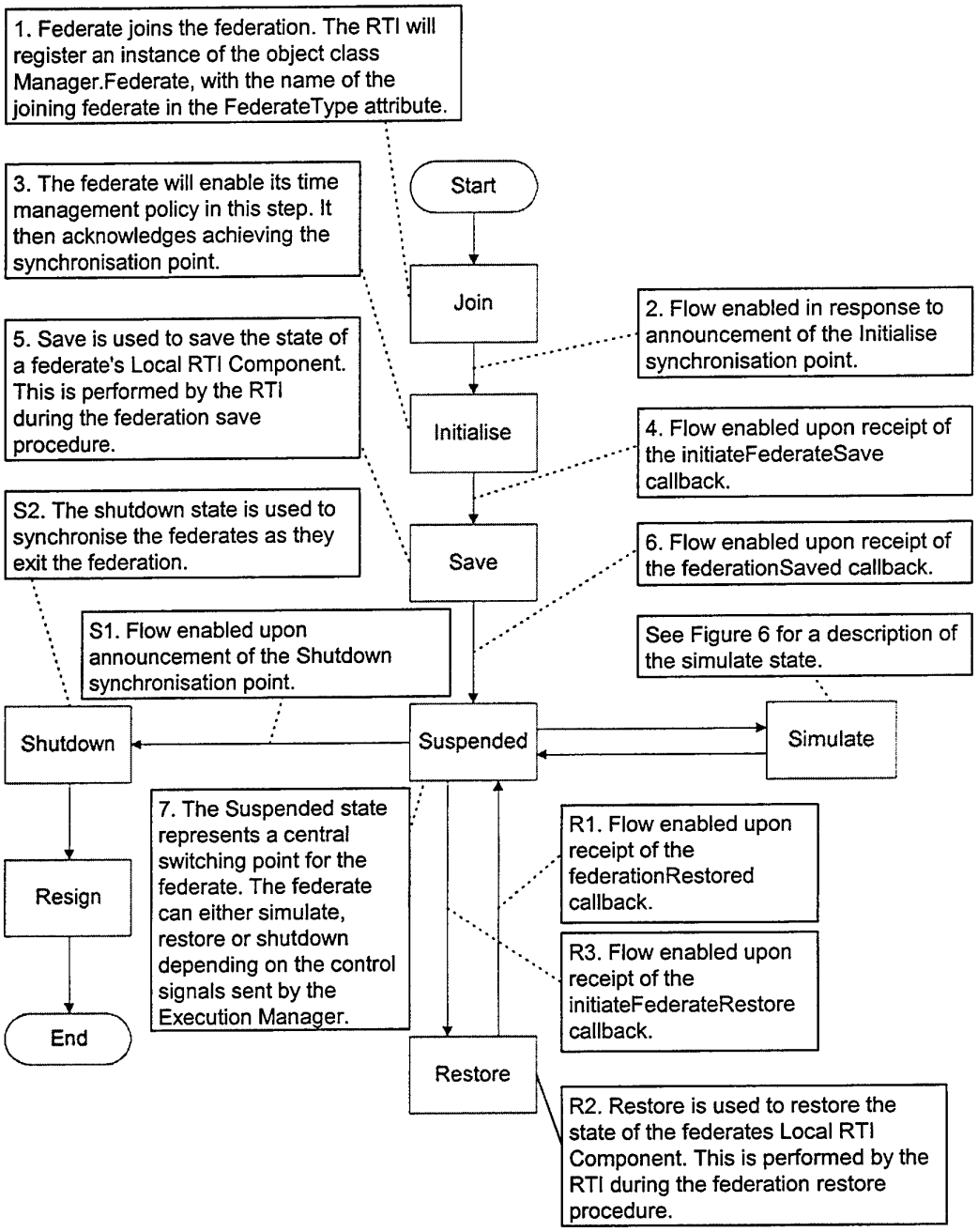

Figure 5: *This flowchart illustrates the flow of an Execution Managed federate, ie, a federate that is subject to the control of the Execution Manager. All flows between states in the flowchart are controlled by the Execution Manager.*

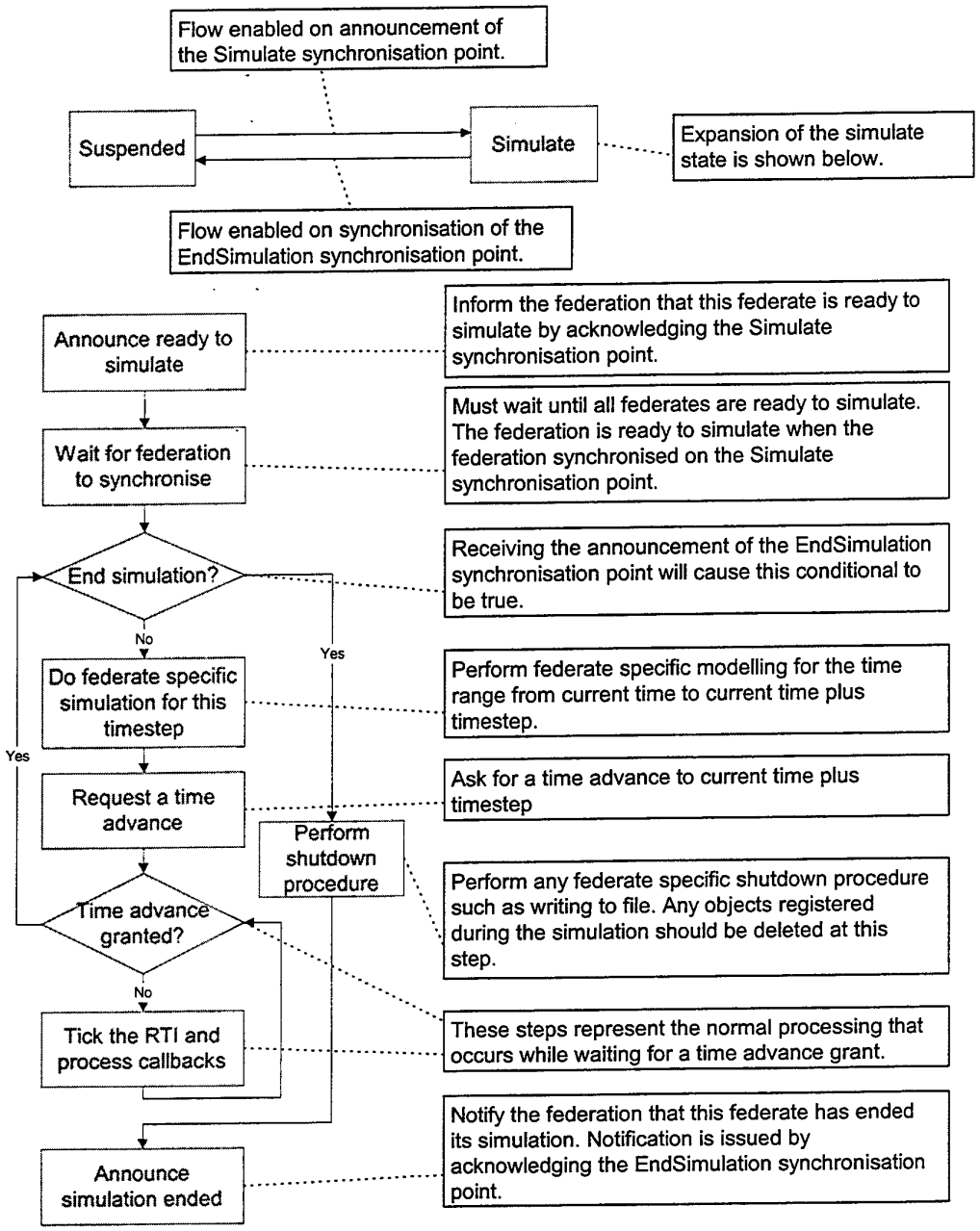

Figure 6: Flow of an Execution Managed Federate : Simulate

3.1.1 Implementation of a participating federate

This section will detail how to implement a participating federate based on the flow charts presented in Figure 5 and Figure 6. Throughout the narrative the participating federate under construction will be referred to as the federate.

After joining a federation, the federate waits for notification to enter the Initialise state. The federate is notified to enter the Initialise state through the reception of the `announceSynchronizationPoint` federate ambassador callback with label "Initialise".

While in the Initialise state, the federate should enable its time management policies and perform any other federate initialisation requirements. Once initialisation is complete, the federate informs the federation of completion of initialisation by calling the RTI ambassador service `synchronizationPointAchieved` with label "Initialise". The federate then waits for notification that the federation has completed the Initialise state. This is communicated through the federate ambassador callback `federationSynchronized` with label "Initialise".

Once the Initialise state is complete the participating federate will wait for the VSEM to move the federation to the Save state by requesting a federation save. This is communicated to the federate through the `initiateFederateSave` federate ambassador callback. The federate will inform that it is saving by first calling the RTI ambassador service `federateSaveBegun`. The federate is then free to perform its federate specific save. Once the save is complete the federate must call `federateSaveComplete`, to indicate a successful save, or `federateSaveNotComplete`, to indicate an unsuccessful save. No RTI ambassador services can be invoked between the calls to `federateSaveBegun` and `federateSaveComplete`/`federateSaveNotComplete`. The federate will then wait until the federation has completed the save. It is informed of this completion through reception of either the `federationSaved` or `federationNotSaved` federate ambassador callbacks.

Upon receiving the `federationSaved` or `federationNotSaved` federate ambassador callbacks the federate is free to enter the Suspended state. From the Suspended state the federate can proceed to three different states (Restore, Simulate or Shutdown) depending on the control message sent by the VSEM.

The federate will move to the Restore state upon receipt of the `federationRestoreBegun` federate ambassador callback and subsequent receipt of the `initiateFederateRestore` federate ambassador callback. The federate must not invoke any RTI ambassador services between reception of the `federationRestoreBegun` and `initiateFederateRestore` federate ambassador callbacks. Upon receiving the `initiateFederateRestore` the federate performs any federate specific restoration. Once complete the federate notifies the RTI that it has finished its restore by invoking the `federateRestoreComplete` or `federateRestoreNotComplete` RTI ambassador service. The federate then waits until the federation has restored which is communicated to the federate via the `federationRestored` or `federationNotRestored` federate ambassador callbacks. Upon receipt of either of these callbacks the federate re-enters the Suspended state.

The federate will move from the Suspended state to the Simulate state upon receipt of the `announceSynchronizationPoint` federate ambassador callback with label "Simulate". The federate will perform any pre-simulation initialisation required and then acknowledge the synchronisation point by calling the `synchronizationPointAchieved` RTI ambassador method with label "Simulate". Once the federation has synchronised on this synchronisation point, communicated to the federate through the federate ambassador callback `federationSynchronized` with label "Simulate", the participating federate can begin its simulation loop.

During the simulation loop the federate may receive execution management interactions. These interactions are shown in Table 2 with the parameters of these

interactions shown in Table 3. See these tables and the associated text description for information on how to process these interactions and parameters.

When handling the ExecutionManagement.CreateCompositeEntity and ExecutionManagement.CreateComponentEntity, initial values for the attributes of the object to be created will be specified. As a minimum these attributes will be drawn from the attributes of the object class that is being created. The federate must be able to provide default values for all attributes defined in the FOM for the appropriate object class but must also be able to process initial values for those attributes requested through the interactions. This requirement is an important one and has led to the specification of the following Virtual Ship Execution Management Rule:

> *An execution managed federate capable of creating an entity shall be able to process the initial attribute data provided to the ExecutionManagement.CreateCompositeEntity or ExecutionManagement.CreateComponentEntity interaction for all attributes in the VS-FOM that are defined for the object class of the entity created by the federate.*

The federate will continue to simulate until it receives notification to end the simulation. This notification is communicated through the `announceSynchronizationPoint` federate ambassador callback with label "End_Simulation". Upon receipt of this callback, the federate will end its simulation loop and perform any necessary post-simulation processing. As part of the post-simulation processing the federate will delete from the RTI any object instances registered by it during the simulation execution. This assures remote federates that all object instances are deleted at the end of a scenario iteration even if they do not receive notification from the RTI of the object deletion. This processing has been formalised in the following Virtual Ship Execution Management Rule:

> *All object instances registered by a federate during a simulation must be deleted from the RTI at the end of a scenario iteration.*

Once post-simulation processing is complete the federate will invoke the `synchronizationPointAchieved` RTI ambassador method with label "End_Simulation". The federate will then wait until the federation has synchronised which is communicated through the `federationSynchronized` federate ambassador callback with label "End_Simulation". Upon receipt of this callback the federate will move back to the Suspended state.

The VSEM will request the federation execution to end by moving the federation from the Suspended state to the Shutdown state. This move is communicated to the federate via the `announceSynchronizationPoint` federate ambassador callback with label "End_Fedex". Upon receipt of this callback the federate will perform any end of federation processing and then acknowledge the synchronisation point by calling `synchronizationPointAchieved` with label "End_Fedex". The federate will then wait for the federation to synchronise which is communicated via the `federation-Synchronised` federate ambassador callback with label "End_Fedex". Upon receipt of this callback the federate is free to resign from the federation execution and then to attempt to destroy the federation execution.

3.2 Auxiliary Federates

In addition to federates that are subject to execution management, a federation may contain auxiliary federates. These are federates that do not respond to execution management.

Although these federates are not explicitly execution managed, the VSEM will wait for them to join along with the rest of the federation before it begins the federation execution.

The current implementation of the VSEM uses the federation management services related to synchronisation points, saves and restores to control the flow of federation execution. Synchronisation points can be registered with only those federates that are identified in the script file as participating so auxiliary federates will never be notified of these. However, federation saves and restores are performed on all federates in the federation. This imposes the restriction on auxiliary federates that they must be able to acknowledge the `initiateFederateSave` and `initiateFederateRestore` federate ambassador callbacks. This requirement may prevent the use of some third party federates within a Virtual Ship federation and is a feature currently under review (see section 5.1).

Auxiliary federates are identified in the script file by using the keyword "AuxiliaryFederate". Identifying a federate as auxiliary tells the VSEM that it should wait for this federate to join but exclude it from any synchronisation points the VSEM registers. See the section 4 for more information on the syntax of the script file.

4 The Script File

A script file dictates the actions of the VSEM. The script contains a list of federates that are to exist in the federation. This list allows the VSEM to determine when all federates have joined. The script contains the end time for a scenario iteration and the number of scenario iterations to execute. This information allows the VSEM to determine when a single scenario iteration is to end and when to end the federation execution. Finally, the script contains a listing of the entities that are to be created during a scenario iteration. The VSEM reads this list and uses the information to request creations at the specified times.

The script file can be thought of as a document for the federation. It uniquely defines a federation execution by listing the federates that exist in the federation execution, and the entities that will be created. It should be noted, however, that the execution management concept does not preclude federates from creating and registering instances of objects not specified in the script file. Eg, a federate modelling fighter aircraft may create an aircraft based on a specification in the script file. The modelling of that created fighter may subsequently result in the creation (launching) of a missile during the execution of the scenario.

A script file is simply a text file written to a specific format. The format is defined in section 4.1 using a modified Backus Naur Form (BNF). Section 4.2 gives an example script file.

4.1 BNF format of script file

This section defines the syntax of a script file in a modified BNF. The script definition is divided into major sections and then given completely in section 4.1.8.

4.1.1 Common items

The script file allows for line comments by beginning the line with a '#'. These comments can exist anywhere in the script file, provided they are on a line by themselves. They cannot be used at the end of lines.

```
comment = "#" string;
```

A `string` element is a standard sequence of ASCII characters without a whitespace. If a string is to contain whitespace, it must be enclosed in quotes (").

```
whitespace_string = quot string quot;
quot = """;
```

For convenience, a non-terminal `strings` is used to allow the possibility of either a string without whitespace (`string`) or a string with whitespace (`whitespace_string`).

```
strings = string | whitespace_string;
```

The simple datatypes `integer` and `double` are used with the usual meaning for each. A non-terminal `decimal` is used to allow either an `integer` or a `double`.

```
decimal = integer | double;
```

Finally, whitespace only holds special meaning in the `whitespace_string` non-terminal. This allows the script files to be tabbed and indented for readability.

4.1.2 Script file

The root element of all script files is `script_file`. This element consists of two subelments, a header element and a simulation element.

```
script_file = header simulation;
```

4.1.3 Header

The header provides the federation construction information. This contains the federation name, a list of participating federates and a list of auxiliary federates. The ordering of the information in this section is not important.

The `participating_federate` rule is used to define those federates that are to be subject to execution management. The `auxiliary_federate` rule is used to identify those federates that will exist in the federation execution but do not participate in execution management.

```
header = "{" federation_name federates* "}";
federation_name = "FederationName = " string ";";
federates = particpating_federate | auxiliary_federate;
participating_federate = "ParticipatingFederate = " string ";";
auxiliary_federate = "AuxiliaryFederate = " string ";";
```

4.1.4 Simulation

The simulation section consists of a single subelement `loop`. A future extension of the VSEM will allow for multiple loop elements to be defined.

```
simulation = "{" loop "}";
```

4.1.5 Loop

The loop element contains all the data relevant to a simulation. It defines how many loops a scenario should be executed, defines the scenario descriptor and random number seed to be used for each iteration, defines the end time for each loop and lists the composite entities that will be created during each iteration.

```
loop = "Loop(" max_loops ")"
             scenario_descriptor
             random_number_seed
             composite_entity*
             end_time
             "EndLoop;";
```

The `max_loops` element identifies the number of scenario iterations to execute.

```
max_loops = integer;
```

The scenario descriptor and random number seed elements are defined as

```
scenario_descriptor="ScenarioDescriptor =" sd_constructed_string ";";
random_number_seed = "RandomNumberSeed= " (integer|loop_index) ";";
sd_constructed_string = strings ("+" strings | loop_index)*;
loop_index = "loop_index";
```

The scenario descriptor can be constructed from a number of strings and the keyword "loop_index". Each item of the constructed string is concatenated using the '+' character. The keyword "loop_index" is used to access the current loop of the simulation. The random number seed can either be a constant integer or the current loop (through the "loop_index" keyword) can be used to change the random number seed at each loop

The time to end a simulation loop is specified by the end_time element.

```
end_time = "EndTime = " decimal ";";
```

4.1.6 CompositeEntity

The composite_entity element specifies the creation of a composite entity in the simulation.

```
composite_entity = "CompositeEntity:"
                           name
                           type
                           time
                           federate
                           config_file
                           attribute*
                           component_entity*
                           "EndCompositeEntity;";
```

The subelements within the composite_entity element define who, how, what and when to create the defined composite entity. A list of component entities that will have this composite entity as a parent are defined as subelements.

The name element identifies the name that should be used when registering the created composite entity with the RTI.

```
name = "Name = " string ";";
```

The type element defines what kind of composite entity is to be created. Valid values for this element are the subclasses of the CompositeEntity object class as specified in the VS-FOM.

```
type = "Type = " string ";";
```

The time element indicates the time at which to create the composite entity. When the Execution Manager's time exceeds this time, the request for creation will be sent.

```
time = "Time = " decimal ";";
```

The federate element identifies which federate is responsible for the creation of this composite entity. For the entity to be created, the federate specified must also be listed as a participating federate in the header section.

```
federate = "Federate = " string ";";
```

The config_file element specifies a configuration file that should be used by the creating federate in the creation of the composite entity. This parameter is a string that may or may not be recognised by the receiving federate. If the receiving federate

does make use of a configuration file and the specified name cannot be found, an execution management error is deemed to have occurred.

```
config_file = "Config_File = " string ";";
```

The `attribute` element defines initial values for attributes that appear in the VS-FOM for the object class being created.

```
attribute = "Attribute : " attribute_name "(" attribute_type_list ") = "
                                                attribute_value ";";
```

The `attribute_name` element is the name of the attribute that an initial value is being supplied for.

```
attribute_name = string;
```

The `attribute_type_list` is used to identify the type of the initial value being supplied. If an attribute is of complex datatype, then the `attribute_type_list` is a comma-delimited sequence of the basic types that appear in the complex datatype.

```
attribute_type_list = attribute_type ("," attribute_type)*;
```

The basic types allowed are defined in the `attribute_type` element. The characters used to define types are:

- `s`: a string type
- `i`: an integer or boolean type (a boolean is defined as a 4-byte integer with zero representing false and non-zero representing true).
- `d`: a double type

```
attribute_type = "s" | "i" | "d";
```

The ordering of the attribute values must match the ordering of the types defined in the `attribute_type_list`.

```
attribute_value = strings | integer | double
                  ("," strings | integer | double )* ";";
```

4.1.7 ComponentEntity

The `component_entity` element is structured similarly to the `composite_entity` element.

```
component_entity = "ComponentEntity:"
                           name
                           type
                           federate
                           config_file
                           attribute*
                           "EndComponentEntity;"
```

The subelements of `component_entity` have the same definition as for the `composite_entity` (section 4.1.6). The interpretation of the `type` element differs in that it defines the type of ComponentEntity to create. This value can be drawn from the fully qualified object class names of subclasses of the ComponentEntity object class specified in the VS-FOM. The attributes that initial values are supplied for are drawn

from the attributes of the ComponentEntity object class and its subclasses as defined in the VS-FOM.

The time that the component entity specified is created at is the same time as its parent composite entity (within which this component entity is defined). Also, the parent composite entity object instance name for this component entity is defined by the name element of the enclosing composite entity.

4.1.8 Complete BNF for a script file

```
script_file = header simulation;

header = "{" federation_name federates* "}";
federation_name = "FederationName = " string ";";
federates = pariticpating_federate | auxiliary_federate;
participating_federate = "ParticipatingFederate = " string ";";
auxiliary_federate = "AuxiliaryFederate = " string ";";

simulation = "{" loop "}";
loop = "Loop(" max_loops ")"
             scenario_descriptor
             random_number_seed
             composite_entity*
             end_time
             "EndLoop;";
max_loops = integer;

scenario_descriptor="ScenarioDescriptor =" sd_constructed_string ";";
random_number_seed = "RandomNumberSeed= " (integer|"loop_index") ";";
sd_constructed_string = string ("+" string | "loop_index")*;
loop_index = "loop_index";

composite_entity = "CompositeEntity:"
                          name
                          type
                          time
                          federate
                          config_file
                          attribute*
                          component_entity*
                          "EndCompositeEntity;";

name = "Name = " string ";";
type = "Type = " string ";";
time = "Time = " decimal ";";
federate = "Federate = " string ";";
config_file = "Config_File = " string ";";

attribute = "Attribute : "
                    attribute_name
                    "(" attribute_type_list ") = "
                    attribute_value ";";
attribute_name = string;
attribute_type_list = attribute_type ("," attribute_type)*;
attribute_type = "s" | "i" | "d";
attribute_value = string | integer | double
                    ("," string | integer | double )* ";";

component_entity = "ComponentEntity:"
                          name
                          type
                          federate
                          config_file
                          attribute*
                          "EndComponentEntity;"
end_time = "EndTime = " decimal ";";
```

4.2 Example script file

This example contains a request to create a sea surface composite entity that contains two sensor component entities, one an electronic support sensor (ESM), the other an infrared search and track sensor (IRST). Each entity is specified as being created by a separate federate and all entities are specified as being created at time = 0.0.

```
# an example script file
header{
FederationName = vsfom;
ParticipatingFederate = Motion;
ParticipatingFederate = ESM;
ParticipatingFederate = GTP_IRST;
AuxiliaryFederate = StealthViewer;
}

simulation{
      #Loop 10 times
      Loop(10)
      ScenarioDescriptor = "Run_" + loop_index;
      RandomNumberSeed = loop_index;

      CompositeEntity:
            Name = VS_01;
            Type = SeaSurface;
            Time = 0.0;
            Federate = Motion;
            Config_File = Motion.cfg;
            Attribute : Position(d,d,d) = 6378137.0,0.0,0.0;
            Attribute : Velocity(d,d,d) = 0.0,0.0,0.0;
            Attribute : Acceleration(d,d,d) = 0.0,0.0,0.0;
            Attribute : Orientation(d,d,d) = 0.0,0.0,0.0;
            Attribute : OrientationRate(d,d,d) = 0.0,0.0,0.0;
            Attribute : Name(s) = SeaSurface.Military.Warship.....;
            Attribute : Description(s) = "A Warship";
            Attribute : Affiliation(i) = 1;
            Attribute : Role(i) = 1;
            Attribute : DRAlgorithm(i) = 6;

            ComponentEntity:
                  Name = VS_01_ESM_01;
                  Type = SensorSystem.ESMSystem;
                  Federate = ESM;
                  Config_File = esm.cfg;
                  Attribute : RelativePosition(d,d,d) = 0.0,0.0,15.0;
                  Attribute : RelativeOrientation(d,d,d)=0.0,0.0,0.0;
                  Attribute:ComponentName(s)=SensorSystem.ESMSystem..GenericESM.Version1_0;
            EndComponentEntity;
            ComponentEntity:
                  Name = VS_01_IRST_01;
                  Type = SensorSystem.IRSystem;
                  Federate = GTP_IRST;
                  Config_File = demo.cfg;
                  Attribute : RelativePosition(d,d,d) = 0.0,0.0,10.0;
                  Attribute : RelativeOrientation(d,d,d)=0.0,0.0,0.0;
                  Attribute:ComponentName(s)=SensorSystem.IRSystem..GTP.Version1_0;
            EndComponentEntity;

            #Add other component entities
      EndCompositeEntity;

      #Add other composite entities
      EndTime = 120.0;
      EndLoop;
}
#End of script
```

5 Future Directions

There are a number of additions and enhancements that have been identified for the Virtual Ship execution management concept.

5.1 Use of 3rd Party federates

Currently the use of 3rd party federates in an execution managed Virtual Ship federation is not possible if those federates do not respond to federation saves and restores. A long-term solution to this problem is to find an alternative to the use of the RTI federation saves and restores. As a short term solution, the concept will be modified so that execution managed federation executions perform saves and restores only if they are to execute for more than one scenario iteration. This allows these 3rd party federates to be used in federation executions having one scenario iteration.

5.2 Other Federate Time Management

The execution manager determines when to end a scenario based on its federate time and an end time specified in the script file. For the other federates in the federation execution to be advised at the correct time that the scenario has come to an end, these federates need to advance time in lockstep with the execution manager. This implies that the other federates need to be time regulating (so the execution manager does not get ahead in time) and time constrained (so that the other federates do not get ahead in time).

The execution management concept is applicable even if federates do not employ time management policies but may require alternative methods for determining the end condition of a scenario iteration. Potential alternatives could be:

- User initiated termination.
- Construct a federate that will end a scenario based on a specific event occurring during the scenario. This allows for non-time based termination to be employed. For example, a federate could be built to keep track of the range between a missile and a ship. Once this range began to increase, indicating that the missile has passed the ship, the federate could terminate the scenario.
- Communicate the end time of a scenario to all participating federates. This will allow each participating federate to proceed at its own pace until the end time is reached.

The current execution management concept also requires that participating federates be time constrained to receive the time stamped ordered creation messages at the correct time.

Federates employing no time management policy may be regarded as being real time, ie, the federate's processing occurs in phase with a clock. A real time federate could execute in an execution managed federation if the concept for execution management were altered to allow the execution manager to execute in real time. Synchronising the clocks the execution manager and the real time federate use will allow for the execution manager and real time federate to step forward in time together. This implies that when the execution manager is at time t, the real time federate will also be at time t. This allows the real time federate to receive time based

end notification at the time intended (subject to network latency), and requests for the real time federate to create entities are assured to be received by the real time federate at the correct time or in its past. If the request is received in the real time federate's past, it can dead reckon the created entity to the federate's current time.

5.3 Use of XML for script files

The format of the script files will eventually be defined in XML. This will allow the script files to be constructed and parsed by a large number of freely available XML tools. This will ease the burden of constructing an execution manager and script files.

5.4 Early termination of scenario iteration/federation execution

The execution management concept will be extended to allow for user requested termination of a scenario iteration before the end of the scenario has been reached. Functionality to terminate a federation execution before all scenario iterations have been performed will also be added. These two functions will initially be available for initiation from the VSEM federate but may be extended to allow early termination to be initiated from any participating federate.

5.5 Dynamic creation of entities

The execution manager issues requests for the creation of entities based on a script file. These creations can be scripted to occur any time during a scenario by specifying the time the request is to be made. The execution manager will be extended to allow dynamic manipulation of a scenario iteration. This will allow a user to dynamically customise a scenario by requesting the creation of entities during the execution of a scenario iteration.

6 Summary

The current specification of the HLA RTI is insufficient for providing high level control of a federation execution. The Virtual Ship concept for execution management addresses this deficiency in a manner appropriate to Virtual Ship federation executions.

This document described the Virtual Ship concept for execution management. The concept defines the high level flow of a Virtual Ship federation execution and this is described in terms of the federation undergoing transitions amongst a finite collection of states. To implement the concept requires three components: a federate known as the Virtual Ship Execution Manager (VSEM), a file input to the VSEM that details how the federation execution should take place and the functionality built into the other federates to facilitate their interaction with the VSEM.

Each of these components has been detailed for the current implementation of the Virtual Ship execution management concept. Future developments of the concept have been identified.

7 References

1. Best, J.P. et al. (2000). *Virtual Ship Architecture Description Document.* DSTO-GD-0257.

8 Acknowledgements

Nicholas Luckman and Jonathan Legg have made significant contributions to development of the concept for execution management.

DISTRIBUTION LIST

Execution Management in the Virtual Ship Architecture - Issue 1.00

Anthony Cramp and John P. Best

AUSTRALIA

DEFENCE ORGANISATION

S&T Program

Chief Defence Scientist
FAS Science Policy
AS Science Corporate Management
} shared copy

Director General Science Policy Development
Counsellor Defence Science, London (Doc Data Sheet)
Counsellor Defence Science, Washington (Doc Data Sheet)
Scientific Adviser to MRDC Thailand (Doc Data Sheet)
Scientific Adviser Policy and Command
Navy Scientific Adviser
Scientific Adviser - Army (Doc Data Sheet and distribution list only)
Air Force Scientific Adviser
Director Trials

Aeronautical and Maritime Research Laboratory

Director

CMOD
RL-MODSA
Dr J. Best (MOD)
A. Cramp (MOD)
Dr J. Legg (SSD)
N. Luckman (WSD)
Dr S. Canney (EWD)
Dr D. Sutton (WSD)
B. Hermans (EWD)
G. Horsfall (EWD)
M. Saunders (WSD)
Dr A. Anvar (MOD)

DSTO Library and Archives

Library Fishermens Bend (Doc Data Sheet)
Library Maribyrnong (Doc Data Sheet)
Library Salisbury (1 copy)
Australian Archives
Library, MOD, Pyrmont
Library, MOD, HMAS Stirling
US Defense Technical Information Center, 2 copies
UK Defence Research Information Centre, 2 copies
Canada Defence Scientific Information Service, 1 copy

NZ Defence Information Centre, 1 copy
National Library of Australia, 1 copy

Capability Development Division
Director General Maritime Development
Director General Land Development (Doc Data Sheet only)
Director General C3I Development (Doc Data Sheet only)
Director General Aerospace Development (Doc Data Sheet only)

Navy
Capability Development Manager, SCFEG, Building 90, Garden Island
SO (Science), COMAUSNAVSURFGRP, Garden Island (Doc Data Sheet only)

Army
ABCA Office, G-1-34, Russell Offices, Canberra (4 copies)
SO (Science), DJFHQ(L), MILPO Enoggera, Queensland 4051
(Doc Data Sheet only)
NAPOC QWG Engineer NBCD c/- DENGRS-A, HQ Engineer Centre Liverpool
Military Area, NSW 2174 (Doc Data Sheet only)

Intelligence Program
DGSTA Defence Intelligence Organisation
Manager, Information Centre, Defence Intelligence Organisation

Corporate Support Program
Library Manager, DLS-Canberra (Doc Data Sheet only)

UNIVERSITIES AND COLLEGES

Australian Defence Force Academy
Library
Head of Aerospace and Mechanical Engineering
Serials Section (M list), Deakin University Library, Geelong, 3217
Senior Librarian, Hargrave Library, Monash University
Librarian, Flinders University

OTHER ORGANISATIONS

NASA (Canberra)
AusInfo

OUTSIDE AUSTRALIA

ABSTRACTING AND INFORMATION ORGANISATIONS
Library, Chemical Abstracts Reference Service
Engineering Societies Library, US
Materials Information, Cambridge Scientific Abstracts, US
Documents Librarian, The Center for Research Libraries, US

INFORMATION EXCHANGE AGREEMENT PARTNERS

Acquisitions Unit, Science Reference and Information Service, UK
Library - Exchange Desk, National Institute of Standards and Technology, US

SPARES (5 copies)

Total number of copies: **56**

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

Execution Management in the Virtual Ship Architecture
Issue 1.00

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Anthony Cramp and John P. Best

5. CORPORATE AUTHOR

Aeronautical and Maritime Research Laboratory
PO Box 4331
Melbourne Vic 3001 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-GD-0258	AR-011-613	General Document	October 2000

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
M9505-19-165	NAV 98/173	DGMD	36	1

13. URL ON THE WORLD WIDE WEB

http://www.dsto.defence.gov.au/corporate/reports/DSTO-GD-0258.pdf

14. RELEASE AUTHORITY

Chief, Maritime Operations Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, SALISBURY, SA 5108

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CASUAL ANNOUNCEMENT Yes

18. DEFTEST DESCRIPTORS

Command and control systems, Computerized simulation, Computer architecture, Naval warfare, Virtual reality

19. ABSTRACT

The Virtual Ship is an application of the High Level Architecture (HLA) in which simulation models that represent the components of a warship are brought together in a distributed manner to create a virtual representation of a warship. It is necessary to impose a degree of control over the flow of execution of these distributed components, beyond that provided by the HLA run-time infrastructure (RTI). This control is referred to as execution management and described in this document.