SCORM 2004 Handbook

Version 1.04

March 2006

The e-Learning Consortium, Japan

Document Revision History

Date	Version	Remarks
December 2005	1.01	Translated from the Japanese document.
February 2006	1.03	Editorial correction
March 2006	1.04	Editorial correction

Table of Contents

1 Introduction

The Sharable Content Object Reference Model (SCORM) has been used in Japan as the standard for Web-based e-learning content and learning system development for over five years. During this time, learning management system (LMS) products, content products and authoring tools that conform to SCORM have been widely used. The SCORM standard that has generally been used so far is SCORM Version 1.2, which was released in 2000. Although a large number of SCORM 1.2 compliant products are widely used, some concerns have been raised as to their functional deficiencies, ambiguities in the specifications and other matters. In 2004, the ADL (Advanced Distributed Learning Initiative) released a new standard called SCORM 2004 in an effort to resolve such concerns and deficiencies in the standard. The purpose of this handbook is to provide a technical overview of this new standard as well as an explanation of the newly added features and the differences from SCORM 1.2. The intended audience of this handbook consists of those who have some knowledge of SCORM.

SCORM 2004 introduces new sequencing and navigation features, and provides an entire range of detailed specifications. The new standard is expected to satisfy most of the requirements expressed by the e-learning industry and the user community. However, the technical content concerning SCORM 2004 exceeds some 800 pages in all, and the big picture cannot be easily understood from the individual technical books. To assist those who have some knowledge of SCORM 1.2 and wish to learn about SCORM 2004, this handbook gives an overview of the SCORM 2004 standard and covers the newly added features and the differences from SCORM 1.2. Readers should find it easier to understand the standard if they read this handbook first and then scan through the standard books.

Section 2 gives an overview of SCORM 2004 with an emphasis on changes from the previous version. Sections 3 and 4 cover the newly introduced sequencing and navigation features. These two sections provide an overview of the new features and explain areas that may be difficult to understand. Section 5 covers the SCORM 2004 Run-Time Environment with an emphasis on the differences from SCORM 1.2. Sections 6 and 7 explain how to implement the sequencing feature and a runtime environment from the standpoint of LMS implementers. Section 8 covers points to be noted when migrating from SCORM 1.2 to SCORM 2004.

The SCORM 2004 technical books and their acronyms that this handbook uses as references are listed below.

OV: *SCORM 2004 2nd Edition Overview*
CAM: *SCORM Content Aggregation Model Version 1.3.1*
RTE: *SCORM Run-Time Environment Version 1.3.1*
SN: *SCORM Sequencing and Navigation Version 1.3.1*
CR: *SCORM Conformance Requirements Version 1.3*
ADD: *SCORM 2004 2nd Edition Addendum Version 1.2*

2 SCORM 2004 Overview

This section describes the significance of the e-learning standard, emphasizing the newly added features in SCORM 2004 and the changes from SCORM Version 1.2.

2.1 What is SCORM?

SCORM is the Sharable Content Object Reference Model documented and maintained by the Advanced Distributed Learning Initiative (ADL) of the United States of America. It is intended to provide a common standard that will enable the sharing of learning content by making it

- Durable
- Interoperable
- Accessible
- Reusable

SCORM is a foundation reference model. E-learning content based on SCORM can be used without change regardless of any changes to the hardware and software environment (durability), can run in any operating system and Web browser environment (interoperability), can be searched for and discovered whenever required (accessibility) and can be used to develop new learning content (reusability).

2.2 Origins of the SCORM Standard

The SCORM standard and specifications are derived from work done by various industry and technology organizations, including the IMS Global Learning Consortium (IMS), the Aviation Industry CBT Committee (AICC), and the Institute of Electrical and Electronics Engineers Learning Technology Standards Committee (IEEE LTSC). Using these specifications and guidelines, SCORM defines a framework for application to learning content, its aggregation, and its packaging. SCORM also defines a set of conformance requirements for systems that will deliver such content to the learner. SCORM has been influenced by the following:

SCORM 2004 CAM

- IEEE Learning Object Metadata (LOM)
- IMS Content Packaging
- IEEE Extensible Markup Language (XML) Schema Binding for Learning Object Metadata Data Model

SCORM 2004 RTE

- IEEE Data Model For Content Object Communication
- IEEE ECMAScript Application Programming Interface for Content to Runtime Services Communication

SCORM 2004 SN

- IMS Simple Sequencing

2.3 The LMS Model

Figure 2.1 shows a generalized learning management system (LMS) model. As shown, an LMS provides various services such as a learner profile service and a content management service. However, SCORM does not address the specific implementation of such services. It provides a set of specifications only for the interface points between learning content and the LMS. It merely defines the rules for registering content to the LMS, launching the content and exchanging data between the content and the LMS.

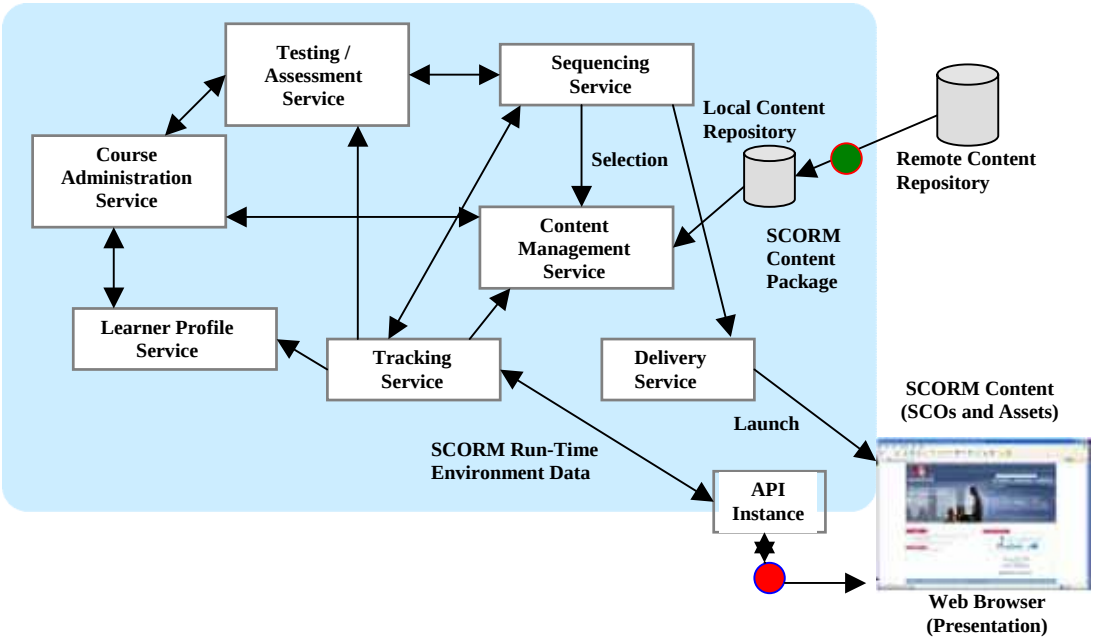


Figure 2.1 An LMS Model

2.4 SCORM 2004 Overview

The ADL released SCORM 2004 in 2004 as the successor standard to SCORM 1.2. SCORM 2004's most significant difference from its predecessor is the addition of sequencing and navigation features. Thanks to the enhancements introduced in the new standard, content developers can enjoy more freedom in their content design and development. For example, the dynamic behavior of content in response to the learning experience sequence and the learner's progress – which could not be described under the earlier versions – can now be controlled by the content. To complement LMS-provided user interface devices for navigation, content can now trigger navigation request events to allow use of the content's own navigation command buttons like [Next] and [Back].

The specifications and standards of SCORM 1.2 were bundled into three technical books: the *SCORM Overview* book, the *SCORM Content Aggregation Model* book and the *SCORM Run-Time Environment* book. For SCORM 2004, the *SCORM Sequencing and Navigation* book has been added to cover the sequencing and navigation specifications, so there are now four technical books.

(1) The SCORM 2004 Overview Book

This book describes the history and objectives of the ADL Initiative and SCORM. It also covers the specifications and standards the SCORM has borrowed, and explains how the SCORM books relate to each other.

(2) The SCORM Content Aggregation Model (CAM) Book

This book covers what content developers should know when they design SCORM compliant learning content; that is, it provides a set of guidelines on how to describe the identity of learning content components, and explains how to assemble and package the components. These guidelines are based on IEEE Learning Object Model (LOM) 1484.12, the AICC guidelines on content structure, IMS Content Packaging and IMS Simple Sequencing.

The major SCORM technical topics covered in this book are sharable content objects (SCOs), assets, content aggregation, content packaging, the package interchange file (PIF), metadata, the manifest file, sequencing and navigation.

(3) The SCORM Run-Time Environment (RTE) Book

This book covers the requirements of the learning management system (LMS) for managing a Web-based Run-Time Environment in terms of launching a learning content object, exchanging data with the content object, and tracking the learner's progress. These guidelines are based on IEEE Application Programming Interface (API) 1484.11.2 and IEEE Data Model 1484.11.1.

The major SCORM technical topics covered in the book include the Application Programming Interface (API), the API Instance, content object launch, session data support methods, and the Run-Time Environment Data Model.

(4) The SCORM Sequencing and Navigation (SN) Book

This technical book has been added for SCORM 2004, and it marks the most significant evolution of SCORM. This book provides guidelines on sequencing; i.e., how to deliver content to the learner. These guidelines are based on the IMS Sequencing Information and Behavior Model. Guidelines on the navigation and user interface devices are also based on this specification.

The major SCORM technical topics covered in this book include activity trees, objectives, sequencing information, navigation information, and the run-time navigation data model.

While each of the SCORM technical books focuses on specific aspects of SCORM, there are some overlapping areas among the books, and those areas are described in such a way that they can be easily referred to from each other.

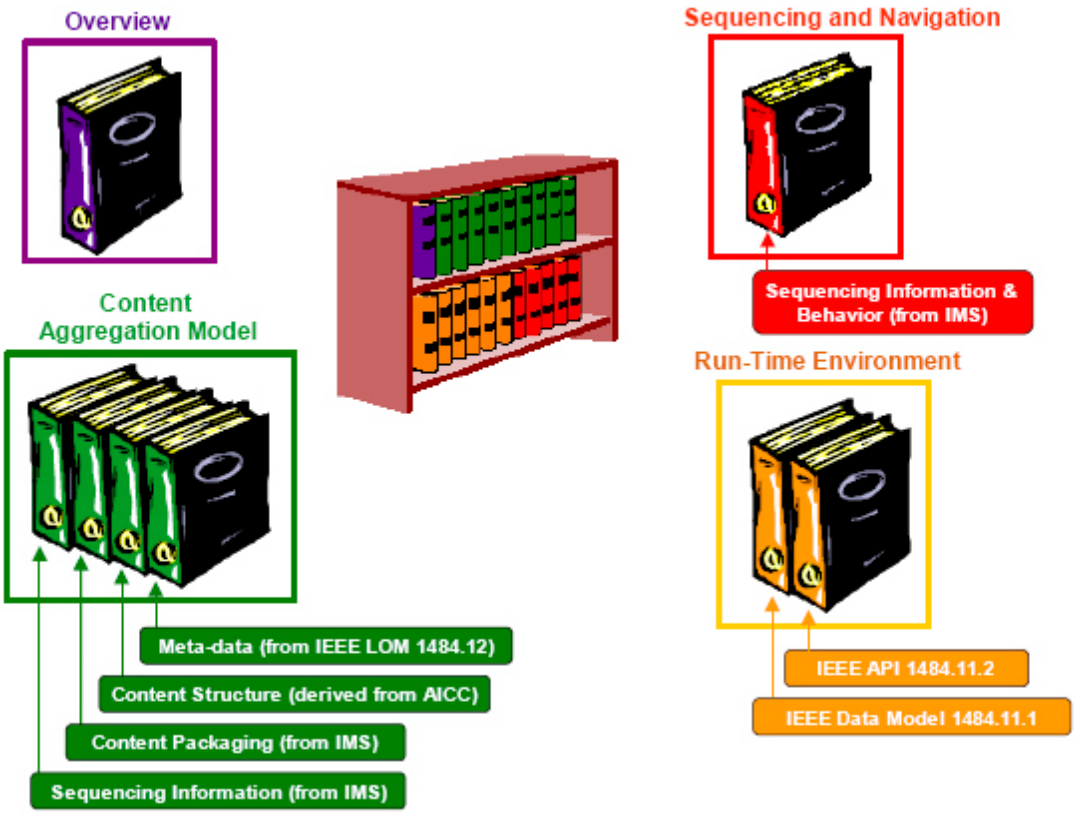


Figure 2.2 SCORM Books (Source: *SCORM 2004 2nd Edition Overview* by ADL)

2.5 Changes from SCORM 1.0 to SCORM 1.2

Many changes have been made in the SCORM specification from the previous SCORM versions. The forces behind this revision include the need to clarify concepts and requirements, changes resulting from standardization efforts, the adoption of best practices from the ADL community, and the provision of enhancements and bug fixes.

SCORM entered its experimentation and evaluation step with SCORM 1.0. The participants in the experimentation and evaluation effort regarding SCORM 1.0 raised a number of questions and issues based on what they encountered during the implementation.

Instead of expanding the scope of SCORM 1.0 in the new version, the specifications and guidelines were modified and improved in SCORM 1.1 on the basis of the feedback from the early participants.

The most noticeable change introduced in SCORM 1.1 was a name change. That is, while SCORM was an acronym for the Sharable Courseware Object Reference Model in SCORM 1.0, it has stood for the Sharable Content Object Reference Model from SCORM 1.1 onwards. This reflects the reality that the standard and specifications in SCORM are applied to various levels of content rather than just courseware as a whole. Another significant change in SCORM 1.1 was that the standards and specifications were subdivided into different functional groups to improve ease of use.

There were two more notable changes introduced in SCORM 1.1. First, a few data model elements were removed from the run-time data model as a result of a simplification introduced to the AICC CMI Recommendations and Guidelines, upon which the SCORM run-time data model was based. Second, a major improvement was made in the API for the Run-Time Environment.

In SCORM 1.2, the SCORM content package application profile was added on the basis of the IMS Content Packaging specifications. In addition, metadata was updated to reflect the latest specifications developed by IMS and IEEE LTSC. This update included changes made to the data model and XML binding specifications. Furthermore, from this version onwards the metadata application profile was renamed the SCORM Content Aggregation Model and a new naming convention was adopted to match the IMS Content Packaging specifications.

2.6 Changes from SCORM 1.2 to SCORM 2004

This section describes the changes and improvements made in SCORM 2004 compared to SCORM 1.2.

2.6.1 Change to the Versioning of SCORM

To improve the independence and maintainability of the SCORM technical books, the versioning of SCORM has been changed as from SCORM 2004 to allow each SCORM book to have its own version, as in CAM and RTE "Version 1.3". Future changes will apply only to those books affected and will be reflected in their version numbers only.

2.6.2 Addition of the Sequencing Feature

For SCORM 2004, sequencing and navigation specifications have been added as a new SCORM technical book. Learning sequence controls were not part of the SCORM specifications up to SCORM 1.2, but they can now be described. For example, the delivery sequence of learning content can now be controlled by changing the type and sequence of the content to be learned based on the result of pretesting done before the start of a lesson. In this case, a learner is regarded as having completed a course after successfully answering questions A and B, while a learner who does not answer successfully is asked to repeat the lesson or repeatedly go through explanation screens before taking the test again; this continues until a learning objective is met.

Thus, it is now possible for content developers to control content behavior by describing a content structure and its associated sequencing rules in the manifest file.

As a learning path or status can be defined by combining various conditions, such as how the learner satisfies an objective and how he or she progresses through the lesson, content developers can now create adaptable content or a simulation package specific to a particular learner.

2.6.3 Triggering Navigation Requests from SCOs

The SCORM 2004 Sequencing and Navigation book introduces a specification regarding the navigation requests that can be made from SCOs.

SCOs are allowed to trigger navigation request events such as [continue] and [previous] from within the SCOs. In addition, SCOs can now request whether LMS-provided navigation user interface (UI) devices are to be shown or hidden.

To enable an SCO to trigger a navigation event, the content developer should add an API method call in the SCO that will set a value to a run-time data model element. The content developer should also add a description in the manifest file regarding how to control LMS-provided UI devices.

The new specification allows content developers to design a navigation function, which is an important content factor, without regard to the type of LMS under which the content will run.

2.6.4 Changes to the SCORM Run-Time Environment

The Run-Time Environment specification has been greatly changed in SCORM 2004 from SCORM 1.2. This section briefly summarizes these changes.

(1) API Instance Name Change

The API Instance has been renamed from API to API_1484_11.

(2) API Method Name Changes

Table 2.1 Changes to API Methods

SCORM1.2	SCORM2004
LMSInitialize(“”)	Initialize(“”)
LMSFinish(“”)	Terminate(“”)
LMSGetValue(parameter)	GetValue(parameter)
LMSSetValue(parameter_1,parameter_2)	SetValue(parameter_1,parameter_2)
LMSCommit(“”)	Commit(“”)
LMSGetLastError()	GetLastError()
LMSGetErrorString(parameter)	GetErrorString(parameter)
LMSGetDiagnostic(parameter)	GetDiagnostic(parameter)

(3) Data Model Changes

The major changes are as follows:

- All data model elements defined in SCORM must be implemented and supported by the LMS.
- The data model has been flattened in SCORM, and the *cmi.core* and *cmi.student_data* elements have been removed.
- The data model element for the interactions between an SCO and its Run-Time Environment has become more sophisticated with the introduction of a more precise format for describing data concerning the learner’s responses to tasks and responses to questions.

- As data model element names and their values are bound to character strings encoded in Unicode (ISO 10646), multi-language implementation has been realized, including double-byte languages.
- The *lesson_status* data model element has been removed, and in its place the *completion_status* and *success_status* elements have been introduced. The possible state token values of the *completion_status* data model element are *completed, incomplete, not_attempted* or *unknown* to indicate whether the learner has completed an SCO. The possible state token values of the *success_status* data model element are *passed, failed* or *unknown* to indicate whether the learner has mastered an objective. The *browsed* state token is no longer used.
- The *score.scaled* data model element has been introduced to indicate the performance level of the learner for an objective. The old *score.raw* data model element with a range from 0 to 100 points has been removed.
- The *objectives* data model element is mapped with the learning objectives of an activity, and shared global objectives can now be defined.
- The introduction of a wider range of error codes makes it possible to check the status of the API Instance and the validity of data.

2.6.5 Changes to the SCORM Content Aggregation Model

The SCORM content aggregation model has been changed to reflect the introduction of sequencing and navigation specifications, and the affected XML schemas have been modified.

In addition, the following ADL content packaging extension elements were removed:

- *<adlcp:prerequisites>*
- *<adlcp:masteryscore>*
- *<adlcp:maxtimeallowed>*

The conditions that were previously defined with these elements are now described in terms of corresponding sequencing rules.

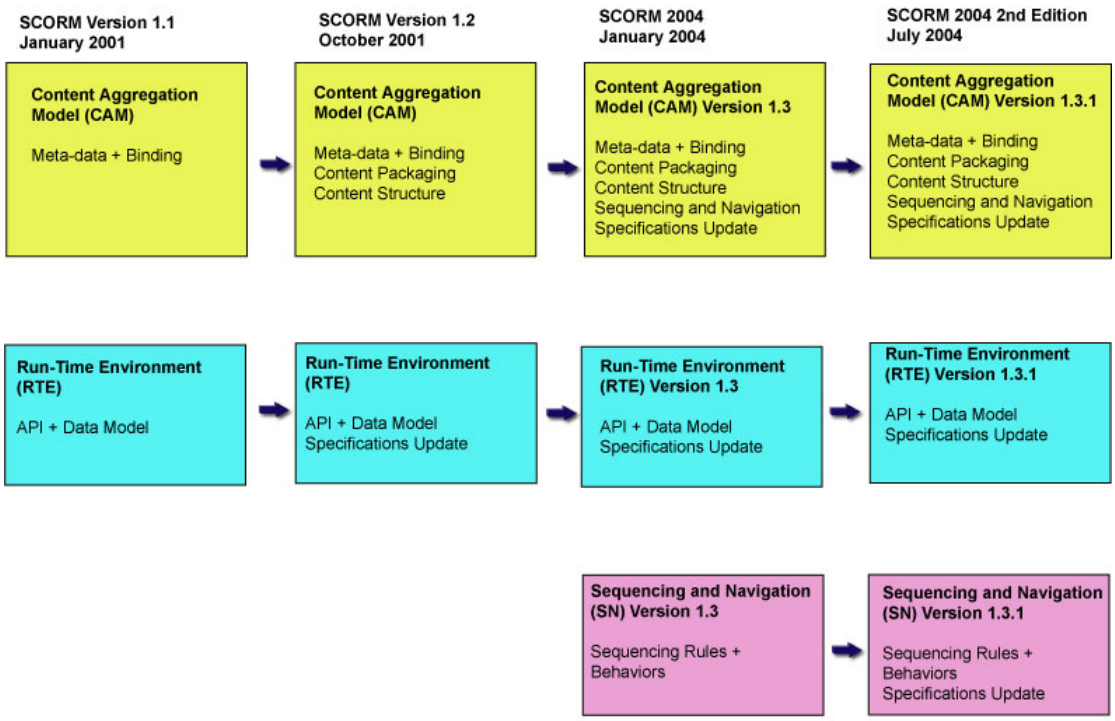


Figure 2.3 SCORM Evolution

2.7 Future Evolution of SCORM

The ADL lists the following as future tasks for Web-based learning functions:

- Designing new run-time and content data model architectures
- Incorporating simulations
- Incorporating electronic performance support objects
- Implementing SCORM-based intelligent tutoring capabilities
- Designing a new content model
- Incorporating gaming technologies

Note, however, that the ADL has no current plans to release the next version.

3 Sequencing

This section explains the sequencing feature that is the main enhancement in SCORM 2004. It covers the basic concept of sequencing and its relationship with other components, and explains how sequencing behaviors are described.

Figure 3.1 shows an overview of a sequencing behavior. Content developers control the behavior of content by describing a content structure and a set of sequencing rules associated with the structure in the manifest file (*imsmanifest.xml*). The LMS reads the manifest file and performs actions described in the file. When it receives a navigation request from the learner, the LMS updates the learner's status data (the tracking data) to reflect the learner's progress status, interprets the sequencing rules, decides on the next content to be delivered, and then delivers it to the learner's environment. The LMS repeats this process until the lesson is terminated.

The major components of sequencing and external functions are described with the following:

- Content structure and learning objectives
- Tracking information
- Navigation requests and sequencing requests
- Sequencing rules

Run-time sequencing behaviors are described as a set of processes shown on the right-hand side of Figure 3.1. The behavior of each process is defined as pseudo-code. Section 6 gives a detailed explanation of the processes and pseudo-code. This section explains the external functions of sequencing with the above four components.

3.1 Content Structures and Learning Objectives

In SCORM 2004, a content structure is described as a hierarchical tree structure. Each node in the tree is called an activity, and the content structure is represented as an activity tree. An activity that has no child activity attached to it is called a leaf activity and is associated with a learning resource (an SCO or asset) that can be delivered to the learner's screen.

When an activity has one or more child activities attached to it, the set is called a cluster. For example, Activities 1.1.3, 1.1.3.1, and 1.1.3.2 in Figure 3.1 constitute a cluster with Activity 1.1.3 as the parent, while Activities 1.1, 1.1.1, 1.1.2, and 1.1.3 constitute another cluster with Activity 1.1 as the parent. A cluster is the basic unit for sequencing behavior, and in many cases the sequencing rules that are defined for a parent activity are applied to the cluster.

Each activity is always assigned with at least one objective, and this is called the primary objective[1]. The role of the primary objective is discussed in the section on

[1] The primary objective is also called a rollup objective. This name is given because it is represented by the PrimaryObjective element in the manifest file. In the SN book, this is called the rollup objective because among the objectives associated with an activity, this objective has the *Objective Contributes*

rollup below. Activities and objectives hold tracking information, which will be discussed in Section 3.2. In addition to the default objective, content developers can associate any number of global objectives to each activity. These global objectives can be shared between multiple activities. Therefore an activity can be associated with multiple shared global objectives in addition to the default objective, and each shared global objective may be shared by multiple activities. A read and write relationship can be defined between a shared global objective and activities. The tracking information status of a shared global objective is determined by the value of the tracking information written from an activity. The activity can read the tracking information of a shared global objective, and refer to it in the application of sequencing rules. Shared global objectives are discussed in detail in Section 3.4.6.

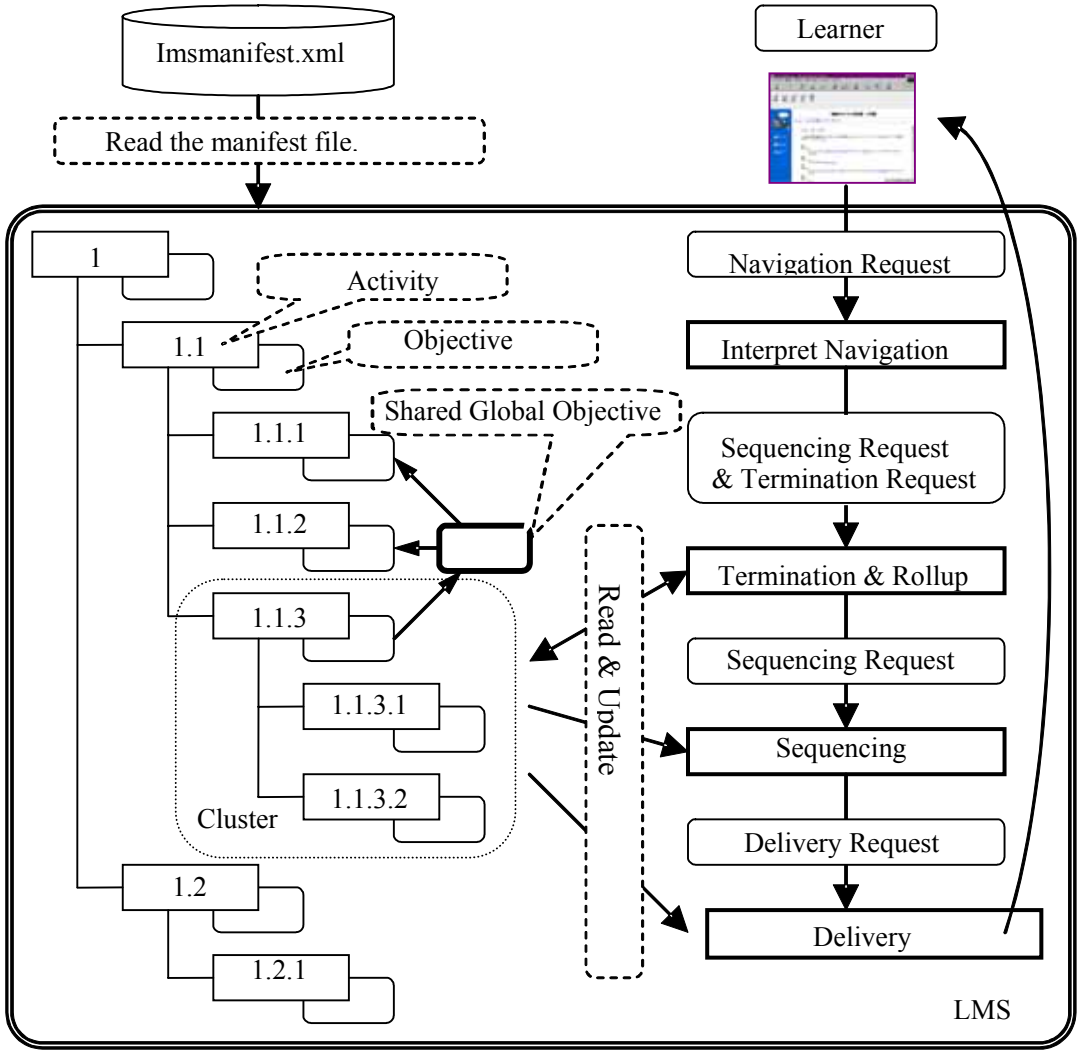


Figure 3.1 Sequencing Behavior Overview

To Rollup attribute set to *True*. They represent the same entity. The term "primary objective" is used in the document.

3.2 Tracking Information

The tracking information is information that reflects the learner's learning status, and it is associated with each activity and its objectives. Table 3.1 shows the details of tracking information.

The tracking information can be divided into data concerning learning performance and completion on one hand and data concerning the learning time and number of attempts on the other. An attempt here means a learner's effort to complete an activity. It refers to an effort from the time when an SCO is launched to the time when it is terminated. It may also refer to a learning effort that starts with a child activity of an intermediate parent activity and continues until the learner navigates out of that cluster. An activity may be attempted more than once, so multiple attempts may exist for an activity. The details of attempts are discussed in 3.5.

Table 3.1 Tracking Model

Objective Progress Information	Activity Progress Information	Attempt Progress Information
Objective Satisfied Status		*Attempt Completion Status*
Objective Normalized Measure		*Attempt Completion Amount*
	Activity Absolute Duration	*Attempt Absolute Duration*
	Activity Experienced Duration	*Attempt Experienced Duration*
	Activity Attempt Count	

3.2.1 Tracking Objective Status and Completion Status

SCORM 2004 makes it possible to manage "completion" and "satisfaction" independently. This is to accommodate a situation where a learner completes an activity from start to finish but does not succeed in attaining its objectives, or a situation where the learner masters the content without completing the learning activity.

"Satisfaction" is related to objectives, indicating whether the learner has satisfied or not satisfied the objective or to what extent the learner has satisfied the objective. These indications correspond to the *Objective Satisfied Status* and the *Object Measure Status* elements of the Tracking Model shown in Table 3.1.

"Completion" is related to attempts on activities. It indicates whether the learner has completed the attempt or not or to what extent the learner has completed the attempt. These indications correspond to the *Attempt Completion Status* and *Attempt Completion Amount* elements of the Tracking Model shown in Table 3.1.

These data elements of a leaf activity are updated by the associated SCO using the Run-Time Environment Data Model. Table 3.2 shows the correspondence between the Tracking Model and the Run-Time Environment Data Model.

In the case of a cluster, however, the status information about the parent activity is updated based on its children's status information. For the whole activity tree, tracking information is propagated from SCO to leaf activity, then its parent, and finally to the root activity of the activity tree. This recursive process is called a rollup[2] behavior. A content developer decides how a parent activity's information is to be updated through a rollup process. Section 3.4.5 describes this rollup process.

Table 3.2 Correspondence between Tracking Model and Run-Time Environment Data Model

Tracking Model		Run-Time Environment Data Model
Attempt Completion Status		*cmi.completion.status*
Attempt Completion Amount		*cmi.progress.measure*
Objective Satisfied Status	Primary Objective	*cmi.success.status*
	Other Objectives	*cmi.objectives.n.status*
Objective Normalized Measure	Primary Objective	*cmi.score.scaled*
	Other Objectives	*cmi.objectives.n.score.scaled*

3.2.2 Information about Learning Time and Attempt Count

The amount of learning time with an activity is managed with the *Attempt Absolute Duration*, *Attempt Experienced Duration*, *Activity Absolute Duration*, and *Activity Experienced Duration* data elements.

The *Attempt Absolute Duration* refers to the duration from the start of one attempt to the end of the attempt. The *Attempt Experienced Duration* refers to the duration from the start of one attempt to the end of the attempt, excluding the elapsed time while the attempt was suspended. If the attempt was not suspended, the values of these two elements are identical.

The *Activity Absolute Duration* refers to the cumulative duration of all attempts on the activity by a specific learner, and the *Activity Experienced Duration* refers to the cumulative duration of all attempts (excluding suspensions) on the activity by a specific learner.

[2] An attempt's completion amount is not subject to rollup in the current specification.

The *Activity Attempt Count* refers to the number of attempts on the activity by a specific learner.

This information is collected at run-time by the LMS.

3.3 Navigation Requests, Sequencing Requests, and Termination Requests

Navigation requests refer to requests like [Continue] and [Previous] invoked by a learner. Table 3.3 shows the types of navigation requests. When a navigation request is made by the learner from the Web browser, LMS-provided user interface devices may be used or a request from the SCO may be honored. Section 5 explains how to issue a navigation request from an SCO.

A navigation request is translated into a termination request and a sequencing request by the navigation interpretation process shown in Figure 3.1. Table 3.3 lists navigation requests and the corresponding sequencing requests. Tables 3.4 and 3.5 explain each of them. A sequencing request triggers the launching of a whole learning process, and traversal from one activity to another. A termination request triggers the termination and suspension of a learning process.

Some sequencing requests and termination requests may be translated into other sequencing requests and termination requests by the post-condition rules shown in Section 3.4.4. The *Retry* sequence request shown in Table 3.4 is generated by the post-condition rules rather than with a navigation request. In response to a sequencing request, the LMS switches from the current activity to another activity and decides on the next activity to be delivered to the learner. In this case, the LMS processes the limit conditions and precondition rules that are described in Sections 3.4.2 and 3.4.3.

Table 3.3 Navigation Requests

Name	Description	Sequence Request	Termination Request
Start	Start a new learning process on the activity tree	*Start*	
Resume All	Resume a suspended learning process on the activity tree	*Resume All*	
Continue	Proceed to the next activity	*Continue*	*Exit*
Previous	Go back to the previous activity	*Previous*	*Exit*
Forward	Not specified in the current version of SCORM		
Backward	Not specified in the current version of SCORM		
Choice	Proceed to the selected activity	*Choice*	*Exit*
Exit	Terminate the current activity	*Exit*	*Exit*
Exit All	Terminate the current activity and all of its ancestors in the tree	*Exit*	*Exit All*
Suspend All	Terminate the current attempt on the current activity and all of its ancestors after saving the tracking information so that the learning process may be resumed in the future	*Exit*	*Suspend All*
Abandon	Abandon the current attempt on the current activity	*Exit*	*Abandon*
Abandon All	Abandon the current activity and all of its ancestors in the tree	*Exit*	*Abandon All*

Table 3.4 Sequencing Requests

Name	Description
Start	Start a new activity
Resume All	Resume a suspended activity
Continue	Proceed to the next activity
Previous	Go back to the previous activity
Choice	Proceed to the selected activity
Exit	Terminate the current activity
Exit All	Terminate the current activity and all of its ancestors in the tree

Table 3.5 Termination Requests

Name	Description
Exit	Terminate the current activity
Exit All	Terminate the current activity and all of its ancestors in the tree
Suspend All	Terminate the current attempt on the current activity and all of its ancestors after saving the tracking information so that the activities may be resumed in the future
Abandon	Abandon the current attempt on the current activity
Abandon All	Abandon the current activity and all of its ancestors in the tree

3.4 Sequencing Rules

Sequencing rules are used by content developers to define sequencing behaviors. The sequencing rules are broadly classified as explained below. Note that these sequencing rules are defined for each activity.

- Rules limiting sequencing requests and transition behavior between activities. These rules are applied on the basis of either a predetermined condition or the tracking information. The former case is called a sequencing control mode. For example, one of the sequencing control modes specifies whether each child activity in an activity tree can be accessed in the forward direction only and reverse navigation is prohibited. The latter type of rules consist of precondition rules, such as "*if the Objective Satisfied Status is True, then the activity is skipped,"* and control condition rules, such as "*the total accumulated learning time for the activity must be less than 30 minutes.*"
- Rules for issuing a specific sequencing request when the tracking information satisfies a certain condition. The rules of this type are called post-condition rules. For example, "*if the activity's objective status is not satisfied, then retry the activity*." Post-condition rules are evaluated at the Termination and Rollup stage shown in Figure 3.1.
- Rules for updating tracking information. As discussed in Section 3.2, the objective progress information for an activity is updated with a trigger event generated when a learner provides some input to the current SCO. The information is rolled up from the activity associated with the SCO through to the root activity. This update process is called the rollup behavior. In these rules, it is possible to describe whether an activity is to contribute to the rollup, conditions under which a rollup occurs, and an appropriate action to be taken when a condition is met. For example, it is possible to describe a rule indicating that "*if three of its child activities are completed, then the parent activity is regarded as completed*." These rules are evaluated at the Termination and Rollup stage shown in Figure 3.1.

The types of sequencing rules discussed above can be summarized from the sequencing behavior standpoint as follows:

(1) Tracking information update
Rollup rules are evaluated at the Termination and Rollup stage shown in Figure 3.1, and the tracking information at each activity of the activity tree is updated.

(2) Confirming a sequencing request
Post-condition rules are evaluated at the Termination and Rollup stage shown in Figure 3.1, and if the post-condition rules are satisfied, the sequencing request based on the navigation request from the learner is replaced with the sequencing request generated by the precondition rule.

(3) Deciding on an activity for delivery
At the Sequencing and Delivery stages shown in Figure 3.1, a target activity is selected for delivery. At this time, the next activity is selected with reference to the sequencing control mode, precondition rules, and the limiting conditions.

Each type of sequencing rule is explained in detail in the sections below.

3.4.1 Sequencing Control Modes

Sequencing control modes control the sequencing behavior for a cluster, and they are classified broadly into the following types:

- Those used to make specific navigation requests effective (*Sequencing Control Choice*, *Sequencing Control Flow*)
- Those to add a limitation to transition between activities (*Sequencing Control Choice Exit, Sequencing Control Forward Only*)
- Those to control how to evaluate tracking information (*Use Current Attempt Objective Information, Use Current Attempt Progress Information*).

Table 3.6 describes these sequencing control modes.

Table 3.6 Sequencing Control Modes

Name	Description
Sequencing Control Choice	A *Choice* navigation request is permitted to target the children of the activity
Sequencing Control Choice Exit	If false, it is prohibited to move from the activity or its descendent to another activity through the *Choice* navigation request
Sequencing Control Flow	Continue and Previous navigation requests are valid in the cluster
Sequencing Control Forward Only	Backward movement is prohibited in the cluster
Use Current Attempt Objective Information	The Objective Progress Information for the current attempt of the activity will be used in rule evaluations and rollup
Use Current Attempt Progress Information	The Attempt Progress Information for the current attempt of the activity will be used in rule evaluations and rollup

3.4.1.1 Sequencing Control Choice and Sequencing Control Flow

The *Sequencing Control Choice* element is used to allow the learner to freely choose an activity from a list of activities when the learner proceeds to the next activity. If the *Sequencing Control Choice* element of a parent activity is set to *True*, then the learner can proceed to any of the child activities with a *Choice* sequencing request. If the sequencing control mode value is *False*, the learner is not allowed to proceed to any of the child activities with a *Choice* sequencing request.

The *Sequencing Control Flow* element is used to decide the delivery sequence of the child activities of the target parent with a *Continue* or *Previous* sequencing request. If the value of the *Sequencing Control Flow* element is *True*, the learner is allowed to proceed forward or backward to each child activity with, respectively, a *Continue* or a *Previous* sequencing request. If the sequencing control mode value is *False*, the learner is not allowed to move within the cluster with a *Continue* or *Previous* sequencing request.

3.4.1.2 Sequencing Control Choice Exit

The *Sequencing Control Choice Exit* element is used to limit movement from the activity or its child activities to another activity with a *Choice* sequencing request. If the value of the *Sequencing Control Choice Exit* element of an activity is *False*, it is not possible to proceed to another activity from that activity or its descendent activities with a *Choice* sequencing request. For a *Choice* sequencing request to be effective, the value of the *Sequencing Control Choice Exit* element defined for the current activity and all its ancestor activities must be *True*. Therefore, this sequencing control mode makes it possible to prohibit an exit from the set of activities below a particular parent activity through a *Choice* sequencing request.

3.4.1.3 Sequencing Forward Only

The *Sequencing Forward Only* element is used to direct movement between activities to forward only within a cluster, and to prohibit reverse movement. If the value of the *Sequencing Control Forward Only* element of an activity is *True*, *Previous* and *Choice* sequencing requests are not allowed among its child activities. However, these sequencing requests are valid if the value is *False*.

3.4.1.4 Use Current Attempt Objective Information and Use Current Attempt Progress Information

These data elements are used to indicate whether the *Attempt Objective Information* and *Attempt Progress Information* for the cluster will reflect only the information gathered from the current attempt or use the latest information, including that from the most recent previous attempt on the cluster's child activities. If the values of these elements of the parent activity are *True*, the information from the current attempt only is reflected. At the current attempt on the associated cluster, the objective information and progress information about the child activities that have not yet been delivered are considered "*Unknown*". If the values of the *Attempt Objective Information* and *Attempt Progress Information* elements are *False*, then the latest information, including that from the most recent previous attempt, is used. At the current attempt on the associated cluster, the objective information and progress information regarding the child activities that have not yet been delivered are adjusted to the information at the end of the most recent previous attempt.

Figure 3.2 illustrates the above process. In Figure 3.2, the *Attempt Objective Information* and *Attempt Progress Information* elements of the parent are set to *True* in a) and *False* in b).

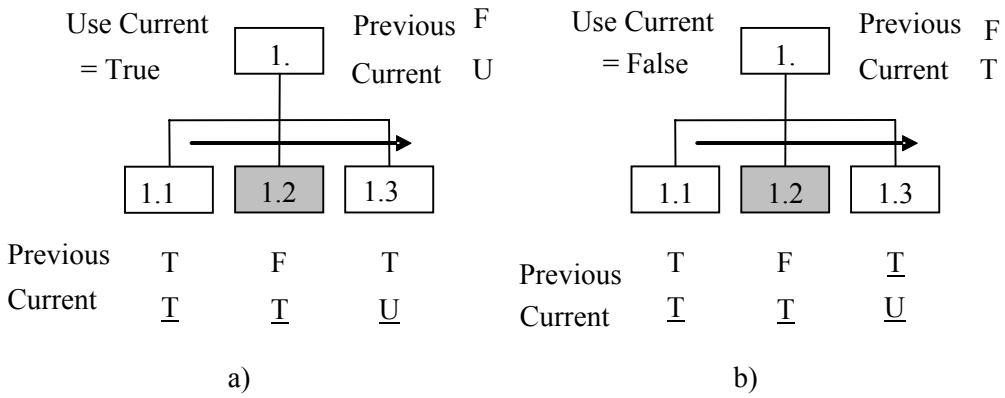


Figure 3.2 Use Current Attempt Objective and Progressive Information

In both cases, the objective satisfied and attempt completion status of the child activities for the previous attempt are set to *True* for 1.1, *False* for 1.2 and *True* for 1.3. In the case of Activity 1, the rolled up status is *False*, based on the default rollup rule explained in Section 3.4.5 for the *And* combination of 1.1, 1.2, and 1.3. At the current attempt, activities 1.1 and 1.2 have been completed with 1.1 and 1.2 being set to *True*.

How should the status information of parent activity 1 be determined in this situation? If it is to be based on the current attempt information, the status of Activity 1 should be set to *Unknown* (as shown in a) above) with the *And* combination of 1.1, 1.2, and 1.3 as Activity 1.3 has not been attempted. If the past information should also be considered, on the other hand, the latest information not only about the current attempt but also about the most recent previous attempt is considered, and the status value of Activity 1 becomes *True* (as shown in b) in the diagram) because the status of 1.3 was *True* at the previous attempt. Content developers may choose which information is used for each cluster.

3.4.2 Limit Conditions

A limit condition can be defined to indicate that an activity is not allowed to be delivered. The only means of implementing a limit condition under the current SCORM specifications is to use the *Limit Condition Attempt Count* element. If an attempt count limit is defined for an activity, that activity cannot be attempted for more than the specified number of times.

3.4.3 Precondition Sequencing Rules

Precondition sequencing rules are used to define conditions for limiting the delivery of activities. These rules are similar to limit conditions in the sense that they are used to control the delivery of activities.

Precondition rules are defined for each activity, and more than one rule may be defined for one activity. The precondition sequencing rules are described in the following format:

If *[condition set]* **Then** [*action*].

The condition set is a collection of conditions to be evaluated as *True* or *False* with respect to the tracking information for the activity. The action part indicates what limit is to be imposed on the delivery of the activity. Some example precondition sequencing rules are shown below:

If Satisfied Then Skip
If the objectives of the activity are satisfied, it is to be skipped.

If Attempted Then Disabled
If the activity has been attempted, the delivery of the activity is to be disabled.

If Always Then Hidden from Choice
At all times, this activity must not be used as a Choice navigation target.

3.4.3.1 Condition Set of Precondition Sequencing Rules

The condition set of a precondition sequencing rule is defined in the following format:

condition_combination ((condition_operator, condition_element), ….)

Therefore, a condition set is one or more pairs of a condition operator and a condition element combined with a condition combination. Each of these elements is explained below.

- Condition combination: There are two condition combination elements: *All* and *Any*. When *All* is used as a condition combination, the condition set is evaluated as *True* only if all the subsequent condition elements are evaluated as *True*. If the *Any* condition combination is used, the result of the condition set becomes *True* if any one condition element is evaluated as *True*. The default condition combination is *Any*.
- Condition operator: There are two types of rule condition operator: *NO-OP* and *Not*. The *NO-OP* condition operator does not change the Boolean value of the corresponding condition element. The *Not* condition operator negates the Boolean value of the condition element.
- Condition element: The evaluation result of a rule condition element depends on the tracking information for the activity. Table 3.7 shows a list of rule condition elements. If the target tracking information is the *Objective Satisfied Status* information or the *Objective Normalized Measure* information regarding the activity, the target objective is specified by the *Rule Condition Referenced Objective* element. For the *Objective Normalized Measure* element, the target threshold is specified by the *Rule Condition Measure Threshold*.

Table 3.7 List of Rule Condition Elements

Condition	Tracking information	Description
Satisfied	*Objective Satisfied Status*	The condition is evaluated as *True* if the *Objective Satisfied Status* for the target objective is *True*
Objective Status Known	*Objective Satisfied Status*	The condition is evaluated as *True* unless the *Objective Satisfied Status* for the target objective is *Unknown*
Objective Measure Known	*Objective Normalized Measure*	The condition is evaluated as *True* unless the *Objective Normalized Measure* for the target activity is *Unknown*
Objective Measure Greater Than	*Objective Normalized Measure*	The condition is evaluated as *True* if the *Objective Normalized Measure* for the target objective is greater than the *Rule Condition Measure Threshold*
Objective Measure Less Than	*Objective Normalized Measure*	The condition is evaluated as *True* if the *Objective Normalized Measure* for the target objective is less than the *Rule Condition Measure Threshold*
Completed	*Attempt Completion Status*	The condition is evaluated as *True* if the *Attempt Completion Status* for the activity is *True*
Activity Progress Known	*Attempt Completion Status*	The condition is evaluated as *True* unless the *Attempt Completion Status* for the activity is *Unknown*
Attempted	*Activity Attempt Count*	The condition is evaluated as *True* if the *Activity Attempt Count* for the activity is 1 or more
Attempt Limit Exceeded	*Activity Attempt Count*	The condition is evaluated as *True* if the *Activity Attempt Count* for the activity is equal to or greater than the *Limit Condition Attempt Limit* for the activity
Always	*None*	The condition is always evaluated as *True*

3.4.3.2 Actions for Precondition Sequencing Rules

Table 3.8 shows a list of actions for precondition rules. These actions are applied when the next activity for delivery is decided as shown in Figure 3.1.

Table 3.8 List of Actions for Precondition Rules

Action	Description
Skip	This action is used for a situation where an activity for delivery is to be selected during a traversal of an activity tree with a *Continue* or *Previous* sequencing request. If the condition for an activity is met, that activity is skipped and the next in line activity in that direction is checked for delivery possibility.
Disabled	This action is used to prohibit the delivery of an activity. If the condition is *True*, that activity cannot be delivered even when selected.
Hidden from Choice	This action is used to hide an activity from the list of available activities for selection with a *Choice* sequencing request. If the condition is *True*, the activity is not included in the target for a *Choice* sequencing request.
Stop Forward Traversal	This action is used when the next activity is to be selected during a forward traversal of an activity tree. If the condition is evaluated as *True*, the activities following it will not be considered candidates for delivery.

3.4.4 Post-Condition Rules and Exit Action Rules

Post-condition rules and exit action rules can be used to ignore a navigation request from the learner and instead generate a sequencing request or termination request designed by the content developer.

These rules are defined for each activity. One or more rules may be defined for one activity. As with the precondition rules, these rules are defined in the form of

If [*condition set*] **Then** [*action*]

The condition set is a collection of conditions to be evaluated as *True* or *False* with respect to the tracking information for the activity. The action part indicates a sequencing request or a termination request.

Two examples of post-condition sequencing rules follow:

If Not Satisfied Then Retry

If the objective of the activity is not satisfied, then that activity is to be retried.

If All (Attempted, Satisfied) Then Exit All

If the activity has been attempted and the objective is satisfied, the whole learning process is to be terminated.

3.4.4.1 Condition Set of Post-Condition Sequencing Rules and Exit Action Rules

The condition part is defined in the same way as for precondition rules.

3.4.4.2 Actions for Post-Condition Sequencing Rules and Exit Action Rules

Table 3.9 shows a list of actions for post-condition rules and exit action rules. These actions are applied when a new sequencing request or a termination request is to be generated in place of the learner's navigation request during the termination and rollup process shown in Figure 3.1.

Table 3.9 Actions for Post-Condition Rules and Exit Action Rules

Action	Description	Sequencing Request	Termination Request
Exit Parent	Terminate the parent of the current activity		*Exit Parent*
Exit All	Terminate the whole learning process		*Exit All*
Exit	Terminate the current activity		*Exit*
Retry	Retry the current activity. If it is not a leaf activity, retry the first child activity of the cluster.	*Retry*	
Retry All	Terminate the whole learning process and resume	*Retry*	*Exit All*
Continue	Move forward	*Continue*	
Previous	Move backward	*Previous*	

3.4.5 Rollup Rules

In the Rollup process, the tracking information of each activity in an activity tree is successively rolled up from each leaf activity (SCO) towards the root activity. Rollup rules are used to decide how the tracking information of a parent activity is rolled up from the tracking information of its child activities.

Rollup rules are related to the *Objective Satisfied Status* data, the *Objective Normalized Measure* data, and the *Attempt Completion Status* data. Figure 3.3 shows the relationship between these three types of data.

In the measure rollup process, the *Objective Normalized Measure* data of the parent's primary objective is determined from the *Objective Normalized Measure* data for its child activities' primary objectives. Only one primary objective is associated with each activity as described in Section 3.1.

The satisfied status of the primary objective of a parent activity is determined through a rollup process using the following data: its own *Objective Normalized Measure* data that is rolled up from its child activities, the *Objective Satisfied Status* data of the

primary objective, the *Attempt Completion Status* data, and the *Activity Attempt Count* data of each child activity.

In the progress status rollup, the *Attempt Completion Status* data of a parent activity is determined from the *Objective Satisfied Status* data, the *Attempt Completion Status* data, and the *Activity Attempt Count* data of each child activity.

Each of these is explained below.

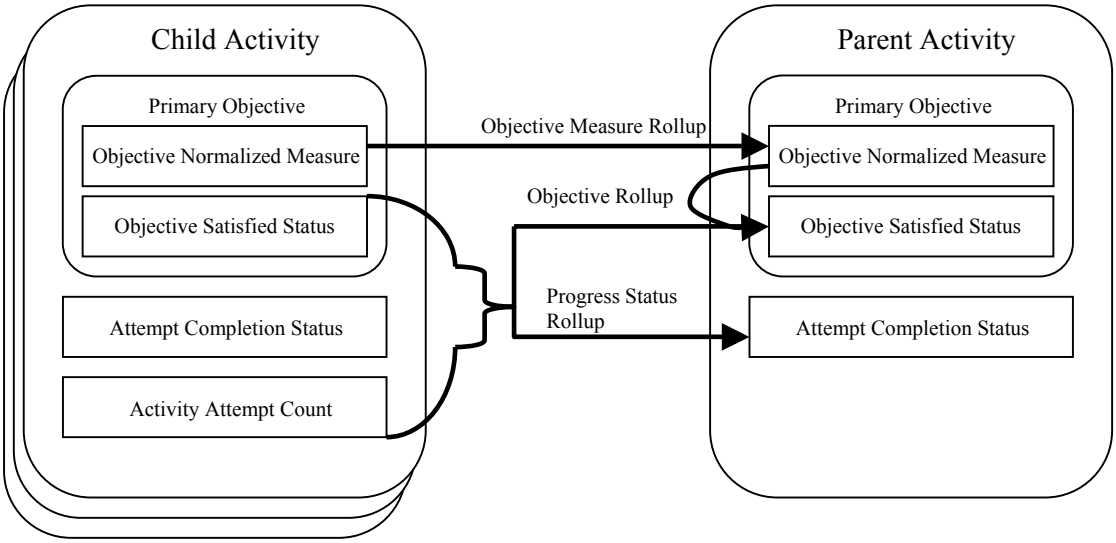


Figure 3.3 Tracking Information Relationship in a Rollup Process

3.4.5.1 Measure Rollup Process

The objective measure of a parent activity's primary objective is determined by calculating the weighted average of the objective measure for the primary objective of each child activity. The weight for an objective measure is specified by the content developer using the *Rollup Objective Measure Weight* data item. The formula is as follows.

$$\text{The Objective Measure of a Parent} = \frac{\sum_{\text{child activity}} (RollupObjectiveMeasureWeight \times ObjectiveNormalizedMeasure)}{\sum_{\text{child activity}} RollupObjectiveMeasureWeight}$$

If the *Objective Normalized Measure* value of a child activity's primary objective is *Unknown*, the calculation is performed as if the value is 0.

3.4.5.2 Objective Rollup

The objective rollup process determines the *Rollup Objective Satisfied* status of a parent activity through the following sequence:

(1) Using the Objective Measure

If the *Objective Satisfied by Measure* element of the parent activity is evaluated as *True*, the *Objective Satisfied Status* element of the parent is determined by comparing its *Objective Normalized Measure* value that has been calculated through a measure rollup process against the value specified for the *Objective Minimum Satisfied Normalized Measure*. If the rollup measure equals or exceeds the value of the *Objective Minimum Satisfied Normalized Measure*, the *Rollup Objective Satisfied* status is *True*; otherwise, it is *False*. The rollup process ends here.

If the *Objective Satisfied by Measure* element of the parent activity is evaluated as *False*, the activity's status does not change and the rollup process proceeds to (2) below.

(2) Using Rollup Rules

If the rollup rules defined for the activity contain an action *Satisfied* or *Not Satisfied*, the satisfied status of the primary objective is determined by evaluating the *Not Satisfied* rule first and then the *Satisfied* rule. Therefore, the result of the *Not Satisfied* rule evaluation may sometimes be overwritten by the result of the *Satisfied* rule evaluation. The objective rollup process ends here. The rollup rules will be discussed in more detail later.

If the actions of the rollup rules for the parent activity do not include *Satisfied* or *Not Satisfied*, the rollup process proceeds to the evaluation of the default rules as shown in (3) below.

(3) Using Default Rules

The following default rules are evaluated in the same way as in (2) above.

If all (attempted or not satisfied), Then not satisfied
If all satisfied, Then satisfied

That is,

If all the activities are attempted or their *Objective Satisfied Status* values are not *True*, the *Rollup Objective Satisfied* status is *False*.
If the *Objective Satisfied Status* values of all child activities are *True*, then the *Rollup Objective Satisfied* status of the parent activity is *True*.

3.4.5.3 Activity Progress Rollup Process

In the progress rollup process, the Rollup Completion Status of a parent activity is evaluated through the following sequence.

(1) Using Rollup Rules

If the rollup rules defined for the activity contain an action *Completed* or *Incomplete***,** the attempt completion status is determined by evaluating the *Incomplete* rule first and then the *Completed* rule. Therefore, the result of the *Incomplete* rule evaluation may sometimes be overwritten by the result of the *Completed* rule evaluation. The progress rollup process ends here. The rollup rules are discussed below in more detail.

If the actions of the rollup rules for the parent activity do not include *Completed* or *Incomplete*, the rollup process proceeds to the evaluation of the default rules as described in (2) below.

(2) Using Default Rules

The following default rules are evaluated in the same way as in (1) above.

If all (attempted or incomplete), Then incomplete
If all completed, Then completed

That is,

If the *Attempt Completed Status* values of all child activities are *True*, then the *Rollup Completion Status* is *True*.
If all the activities are attempted or their *Attempt Completion Status* values are *True*, the *Rollup Completion Status* is *True*.

3.4.5.4 Rollup Rules in Detail

The *Satisfied* and *Not Satisfied* rollup rules for an objective rollup and the *Completed* and *Incomplete* rollup rules for an activity progress rollup process are both defined in the following format:

If [*condition_set*] **For** [*child_activity_set*] **Then** [*action*]

The above rule format means that each rollup rule consists of a *child_activity_set,* which is a set of child activities to consider, a *condition_set*, which is a set of conditions that are to be evaluated against the tracking information of the included child activities, and an *action,* which is a corresponding action that sets the cluster's tracking status information if the final result of applying the set of conditions to the child activity set is evaluated as *True*.

Examples of the rollup rules are shown below.

If not satisfied For any Then not satisfied
If any child activity is not satisfied, then the cluster is not satisfied.

If satisfied For at least 3 Then satisfied
If at least three child activities are satisfied, then the cluster is satisfied.

If satisfied or completed For all Then completed
If all child activities are satisfied or completed, then the cluster is completed.

If satisfied and attempted For all Then satisfied
If all child activities are satisfied and attempted, then the cluster is satisfied.

If not attempted For at least 50% Then incomplete
If at least 50% of the child activities are not attempted, then the cluster is incomplete.

►Rollup Condition Set

A condition set of a rollup rule is defined in the following format:

condition_combination ((condition_operator, condition_element), ….)

This is in the same format as for precondition sequencing rules (Section 3.4.3.) The two rule condition combination elements (All and Any) and the two rule condition operators (Not and NO-OP) are the same as those in precondition rules.

The rule condition elements of a rollup rule differ from those of a precondition or post-condition sequencing rule. Table 3.10 shows a list of condition elements used in rollup rules. The differences from those of a precondition or post-condition sequencing rule are that, first, there is no condition element for comparing the values of objective measures, and second, as the target objective is limited only to the primary objective value, no objective is specified.

►Rollup Child Activity Set

The *Rollup Child Activity Set* element is used in a rollup rule to define how to determine the final evaluation result of *True* or *False* from the application of the specified condition set to the child activities. For example, a rollup rule can be defined specifying that if 80% of the child activities satisfy the specified condition set, then the final result is set to *True*. Table 3.11 shows a list of rollup child activity elements.

Table 3.10 List of Rollup Rule Condition Elements

Condition	Tracking Information	Description
Satisfied	*Objective Satisfied Status*	The condition is evaluated as *True* if the *Objective Satisfied Status* for the primary objective is *True*
Objective Status Known	*Objective Satisfied Status*	The condition is evaluated as *True* unless the *Objective Satisfied Status* for the primary objective is *Unknown*
Objective Measure Known	*Objective Normalized Measure*	The condition is evaluated as *True* unless the *Objective Normalized Measure* for the primary objective is *Unknown*
Completed	*Attempt Completion Status*	The condition is evaluated as *True* if the *Attempt Completion Status* for the activity is *True*
Activity Progress Known	*Attempt Completion Status*	The condition is evaluated as *True* unless the *Attempt Completion Status* for the activity is *Unknown*
Attempted	*Activity Attempt Count*	The condition is evaluated as *True* if the *Activity Attempt Count* for the activity is 1 or more

Condition	Tracking Information	Description
Attempt Limit Exceeded	*Activity Attempt Count*	The condition is evaluated as *True* if the *Activity Attempt Count* for the activity is equal to or greater than the *Limit Condition Attempt Limit* for the activity
Never	*None*	The condition is always evaluated as *False*

Table 3.11 List of Child Activity Sets

Name	Description
All	If the condition combination of all child activities is evaluated as *True*, then the specified rollup action is applied
Any	If the condition combination of any child activity is evaluated as *True*, then the specified rollup action is applied
None	If none of the child activities contains a condition combination that is evaluated as *True*, then the specified rollup action is applied
At Least Count	If the number of activities that contain a condition combination evaluated as *True* is at least equal to the number specified by the *Rollup Minimum Count* element, then the specified rollup action is applied
At Least Percent	If the percentage of activities containing a condition combination evaluated as *True* is at least equal to the number specified by the *Rollup Minimum Percent* element, then the specified rollup action is applied

How to specify a child activity set which is to be subject to a rollup rule is discussed here. In general, the tracking status data for all child activities is used in the rollup evaluations of the parent activity. However, a content developer can selectively include or exclude certain activities from the rollup child activity set for rollup evaluations. For example, when the *At Least Count* element is used for a rollup activity child set definition in a rollup rule, the number of child activities containing a condition combination evaluated as *True* can be calculated by excluding child activities from the target rollup child activity set under the following circumstances:

- A child activity whose *Tracked* element is defined as *False*, which means that no tracking status data is maintained for the activity, is never included during rollup.
- A child activity whose *Rollup Objective Satisfied* element is defined as *False* is not included in the evaluation of rollup rules having a *Satisfied* or *Not Satisfied* rollup action.
- A child activity whose *Rollup Progress Completion* element is defined as *False* is not included in the evaluation of rollup rules having a *Completed* or *Incomplete* rollup action.
- A child activity that has various *Required for* rollup elements defined, which indicate, conditionally, when an activity is included in the evaluation of rollup

rules having specified rollup actions, might not be included. *Required for* rollup elements are shown in Table 3.12.

Table 3.12 List of Required for Rollup Elements

Name	Description	Vocabulary (common)
Required for Satisfied	Indicates when the action is considered in the *Satisfied* rollup rule	• *always* – To be always considered. • *ifNotSuspended* – To be considered when the child activity has been attempted but is not suspended at the time of evaluation • *ifAttempted* – To be considered when the child activity has been attempted at the time of evaluation • *ifNotSkipped* – To be considered if the child activity has not been skipped at the time of evaluation
Required for Not Satisfied	Indicates when the action is considered in the *Not Satisfied* rollup rule	
Required for Completed	Indicates when the action is considered in the *Completed* rollup rule	
Required for Incomplete	Indicates when the action is considered in the *Incomplete* rollup rule	

►Rollup Rule Actions

The *Rollup Rule Action* element specifies one of four actions, *Satisfied, Not Satisfied, Completed*, or *Incomplete*, that will be applied to the parent activity to which rollup rule is associated. Rollup Rule Actions are shown in Table 3.13.

Table 3.13 List of Rollup Actions

Rollup action	Description
Satisfied	The *Objective Satisfied Status* of the rollup objective for the associated parent activity is set to *True*
Not Satisfied	The *Objective Satisfied Status* of the rollup objective for the associated parent activity is set to *False*
Completed	The *Attempt Completion Status* of the rollup objective for the associated parent activity is set to *True*
Incomplete	The *Attempt Completion Status* of the rollup objective for the associated parent activity is set to *False*

3.4.6 Local Objectives and Shared Global Objectives

Each activity is associated with one local objective by default. Moreover, a course developer can specify any number of local objectives for a given activity as required.

These local objectives can be associated with shared global objectives. The shared global objectives enable activities to share tracking information for sequencing. The introduction of global objectives makes it possible, for example, for the content developer to define a situation where a learner is led through a tutorial activity or can skip it depending on the result of a pretest activity. Figure 3.4 shows the relationships between activities, local objectives and shared global objectives.

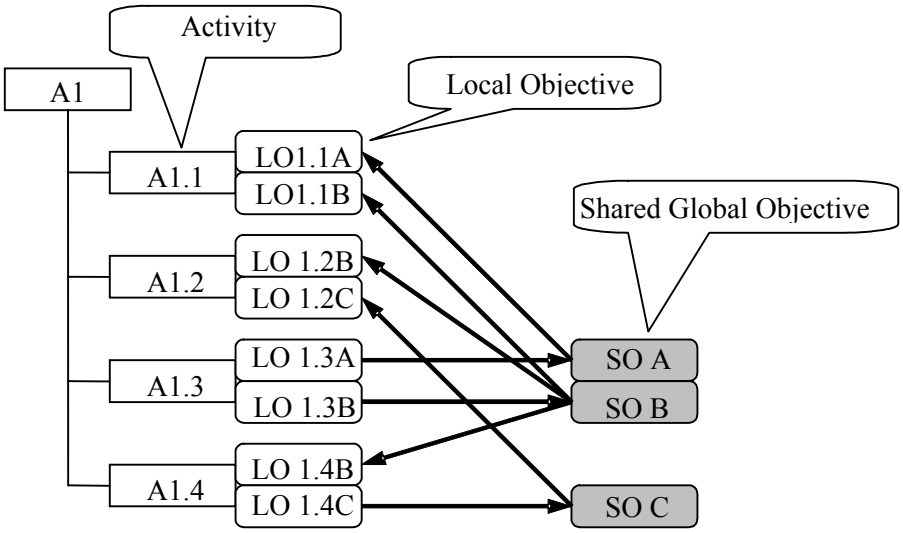


Figure 3.4 Relationships between Activities, Local Objectives and Shared Global Objectives

There are limitations, though, on how activities are related to local and shared global objectives:

- An activity may have more than one local objective. For example, refer to the relationship of A1.1 with LO1.1A and LO1.1B in Figure 3.4.
- One local objective can be related to only one shared global objective. For example, refer to the relationship between LO1.1A and SO A.
- One shared global objective can be related to more than one local objective. For example, refer to the relationship of SO B with LO1.1B, LO1.2B, and LO1.4B.
- The above means that one activity can be related to shared global objectives through the activity's associated local objectives. For example, refer to the relationship of A1.1 with SO A and SO B.
- From the perspective of a shared global objective, one shared global objective can be related to multiple activities through its connection with the local objectives of those activities. For example, refer to the relationship of SO B with A1.1, A1.2, and A1.4.

When a local objective is connected to a shared global objective, the direction for transmission of the objective measure and satisfaction status data should be defined. That is, whether the local objective data is to be written to the shared global objective data or the shared global objective data is to be read to a connected local objective should be defined. The following restriction is applied in this situation:

- For a given activity and a shared global objective, the information from only one local objective can be written to that shared global objective, and the shared global objective data cannot reflect information from any other local objective. For example, refer to the relationship of SO B with LO1.1B, LO1.2B, and LO1.4B in Figure 3.4.

3.4.7 Shared Global Objectives and Rule Evaluations

3.4.7.1 Shared Global Objectives with Precondition, Post-Condition and Exit Rules

As discussed in 3.4.3 Precondition Sequencing Rules and 3.4.4. Post-Condition Rules and Exit Action Rules, the condition part of these rules can refer to the local objectives associated with the activity. If the connection between a local objective of the activity and a shared global objective is defined in such a way that the information from the shared global objective is to be read into the local objective, the information from the shared global objective is used in the rule evaluation.

3.4.7.2 Shared Global Objectives and Rollup Rules

As described in Section 3.4.5 Rollup Rules, only the primary objective of an activity is used for rollup. If the primary objective of an activity is connected to a shared global activity, the shared global objective information is affected by the rollup rules.

Figure 3.5 shows the relationship between rollup and shared global objectives. Rollup rules are evaluated on the basis of each leaf activity whose tracking information has changed because of a state transition on the associated SCO and the activities which refer to the shared global objective whose value is written from the leaf activity. The collection of activities that is the base for rollup is called a rollup set. If there is a change to the tracking information of Activity A1.1.1 in Figure 3.5, for example, the rollup set includes three activities: A1.1.2 and A.1.2, which read the value reflecting the state change from the shared global objective SO B, and A.1.1.1.

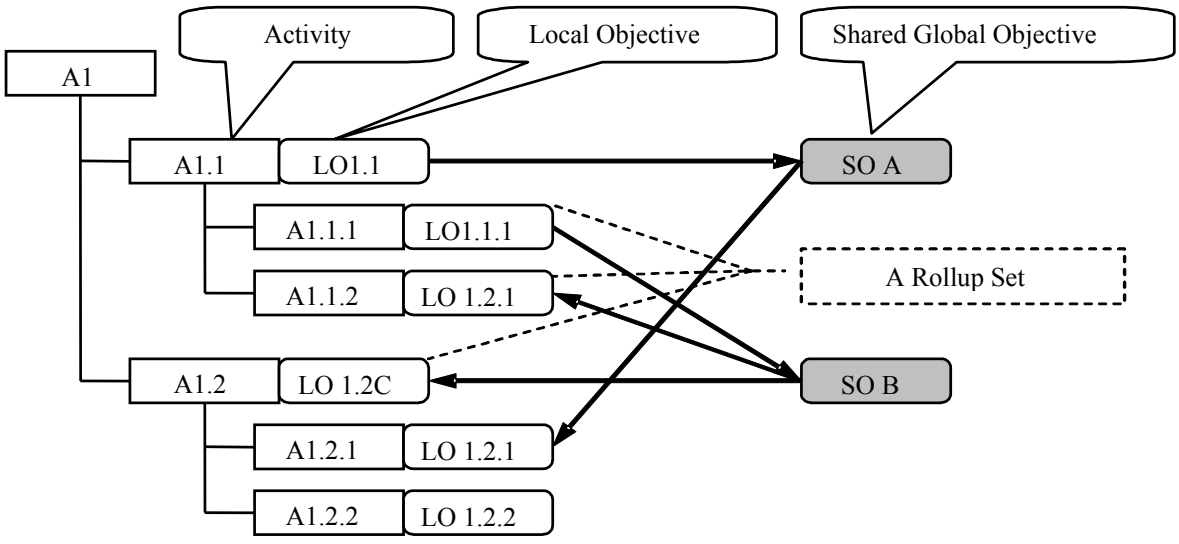


Figure 3.5 Relationships between Shared Global Objectives and Rollup

If a rollup process originating from an activity in the rollup set reaches another activity in a rollup set, the second activity is removed from the rollup set. Furthermore, if the information regarding a shared global objective is updated during a rollup process to reflect a change in the state of an associated local objective of an activity,

no separate rollup behavior occurs among the other activities that refer to that shared global objective. In Figure 3.5, for example, although the tracking information of A1.1 is affected by the rollup behavior from A.1.1.1 and the shared global objective SO A reflects this change, the change is not propagated to A1.2.1.

3.5 Attempts

An attempt refers to a learning effort from the point at which an activity is delivered to the learner and the learner starts the activity to the point at which the learner completes the activity and another activity is selected for delivery. If an activity has been completed during an attempt and that activity is selected and delivered again, this is regarded as a separate attempt.

Attempts are managed in terms of parent and child relationships in an activity tree. Therefore, if A1.1.2 in Figure 3.6 is being currently attempted, for example, A1.1.2, A1.1, and A1 are all regarded as being attempted. If a learner has selected A1.1.1 after having completed A1.1.2 in this case, A1.1.2 is regarded as completed but A1.1 and A1 are still regarded as being attempted.

An attempt can be suspended by a Suspend All navigation request. An attempt on an activity can be resumed from the suspended state through a Resume All navigation request. A resumed attempt is not regarded as a new attempt, but as a continuation of the suspended attempt.

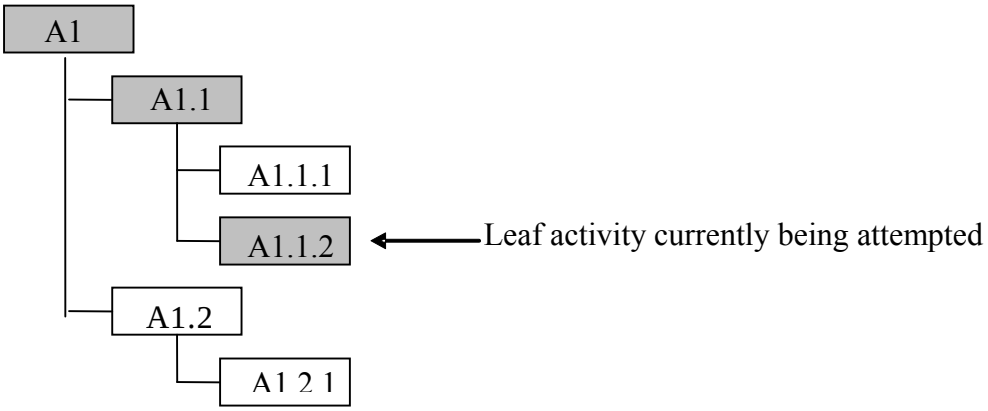


Figure 3.6 Attempts

4 Navigation

This section explains the navigation feature that has been added to SCORM 2004. It covers the basic navigation concepts from a broad perspective, implications of the navigation feature for user interface implementation, and how navigation behavior can be specified.

4.1 Navigation Control Overview

4.1.1 SCO Navigation in SCORM 1.2

The specifications of SCORM 1.2 stipulate that all navigation controls between SCOs should be provided by the LMS. For example, the user interface controls required for the presentation of an SCO and the transition from that SCO to another SCO must be provided by the LMS under SCORM 1.2. In other words, content developers had no say on the navigation behavior between SCOs, and thus could not provide navigation buttons from one SCO to another.

In addition, SCORM 1.2 did not define how an LMS should manage the navigation behavior of SCOs. This made it difficult to provide consistent user interfaces since the user interfaces differed, depending on each LMS, with respect to the presence or absence of buttons and menus and their display positions on the screen, as well as in their captioning and navigation methods. That is, these limitations on navigation control design hindered content developers creating content for multiple LMS environments under SCORM 1.2 who wanted to design and provide user interfaces with a consistent look and feel.

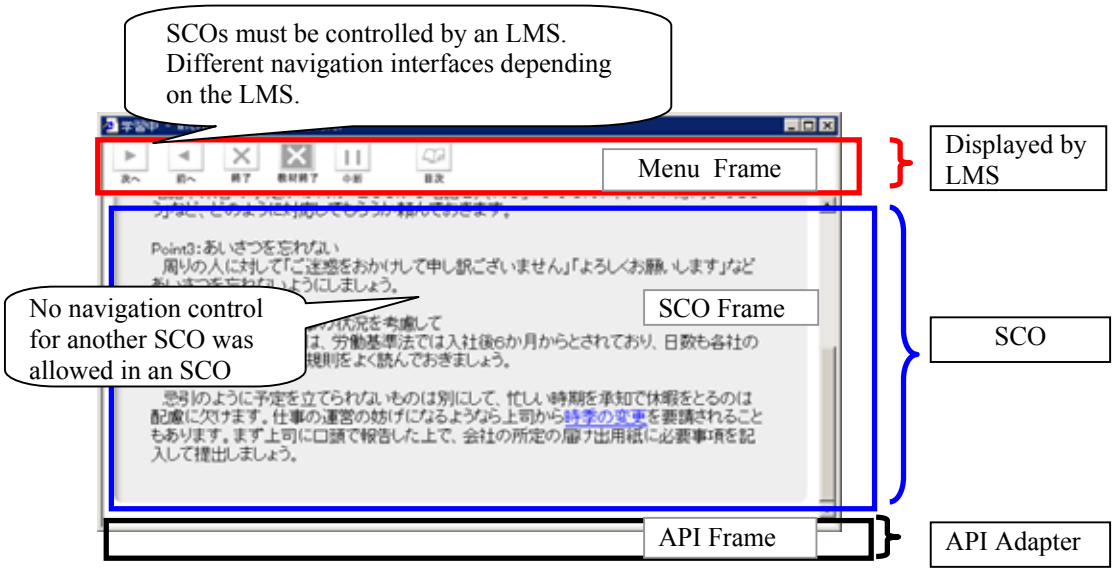


Figure 4.1 Example of SCOM 1.2 Navigation Implementation

4.1.2 SCO Navigation in SCORM 2004

In SCORM 2004, a set of new specifications has been added for managing the navigation methods of SCOs and content developers can now control the navigation of SCOs. More specifically, SCOs can now issue SCO navigation requests and request that the LMS display or hide navigation buttons.

This new functionality enables content developers to standardize a consistent navigation design, considered an important goal in content development, without taking into account the type of LMS environment in which the content will be used.

Note that the LMS does not interfere with the navigation control within an SCO (or an asset) under SCORM 2004 or SCORM 1.2, and the content developer must consider all aspects of this.

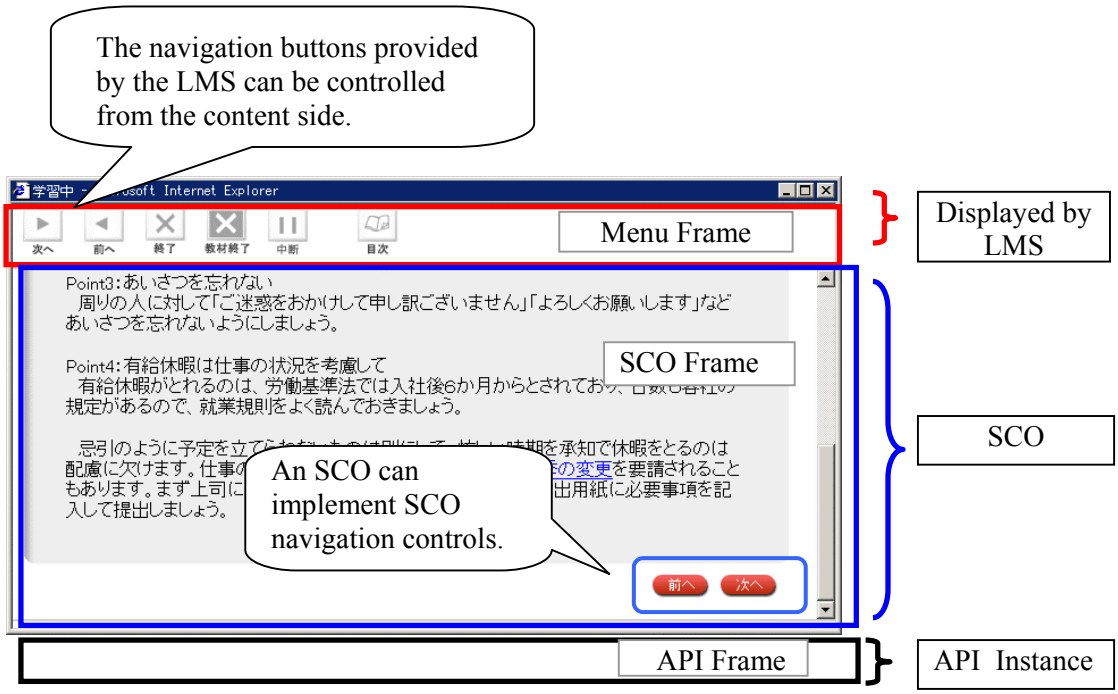


Figure 4.2 Example of SCORM 2004 Navigation Implementation

4.2 Triggering a Navigation Event and SCO Termination

4.2.1 SCO Navigation Event Triggered by an SCO

In SCORM 2004, a set of navigation events like Continue and Previous can be triggered from an SCO. These events are now available in addition to the SCO navigation requests issued by an LMS that could be used in previous SCORM versions. The navigation events triggered by an SCO are processed in the same way as those triggered by an LMS.

The navigation events that can be triggered from an SCO are listed in Table 4.1.

Table 4.1 Navigation Events that can be Triggered from an SCO

Navigation event	Behavior description
continue	This event leads to the termination of the current SCO and the issuing of a *continue* navigation request
previous	This event leads to the termination of the current SCO and the issuing of a *previous* navigation request
choice	This event leads to the termination of the current SCO and the issuing of a *choice* navigation request
exit	This event leads to the ending of the current attempt on the current activity, and to the issuing of an *exit* navigation request
exitAll	This event leads to the ending of the current attempt on the current activity tree and all the associated activities, and to the issuing of an *exit all* navigation request
abandon	This event leads to the ending of the current attempt on the current activity, and to the issuing of an *abandon* navigation request
abandonAll	This event leads to the ending of the current attempt on the current activity tree and all the associated activities, and to the issuing of an *abandon all* navigation request
*_none_*	This event leads to the clearing of all navigation requests that are not yet processed

The navigation requests that are generated by navigation events triggered by an SCO are considered valid for processing in conjunction with sequencing control modes in the same way as SCO navigation requests issued by an LMS. A sequencing control mode specified on a cluster defines the navigation requests that can be applied to the cluster's child activities.

If the *Sequencing Control Choice* is valid, for example, a *choice* navigation request is applied to the child activities of the cluster. In the same way, if the *Sequencing Control Flow* is valid, a *continue* or *previous* navigation request can be applied to the child activities of the cluster.

4.2.2 Navigation Request Event and SCO Termination

To allow an SCO to trigger a navigation event, an element called *adl.nav.request* has been introduced in the navigation data model of SCORM 2004. An SCO triggers a navigation event in the runtime environment or in the LMS by calling a SetValue function with a value like *continue*, *previous*, *choice* or *exit* for the *adl.nav.request* navigation data model element in the following format:

`SetValue("adl.nav.request",`<REQUEST>`)`
where <REQUEST> is one of the following: *continue, previous, choice, exit, exitAll, abandon,* or *abandonAll*.

Note that to trigger a **choice** navigation event, it is necessary to specify the identifier of an activity to be delivered. The format is as follows:

`SetValue("adl.nav.request", "{target =<STRING>}choice")`
where <STRING> indicates the item identifier of the target activity.

When an SCO has communicated a navigation request using an API method call to the LMS with a navigation request event, the LMS performs a sequencing process based on the navigation request after having accepted the termination processing of the SCO. Even if a navigation request has been communicated to the LMS, the LMS does not immediately respond to it, but starts processing the request only when it has accepted a termination request from the SCO.

An SCO may repeatedly invoke the function for a navigation request event many times, but each time it invokes that function the value set by the SCO is replaced with the new value provided in the latest function call. This means that the only event to be responded to is the last navigation request event that is recorded after the LMS has completed processing a terminate request from the SCO.

Figure 4.3 Example of Using a "continue" Navigation Event

4.2.3 Validity of Navigation Request Events

To enable SCOs to check with the LMS as to whether a navigation request *continue*, *previous*, or *choice* is valid, a new element (*adl.nav.request_valid.*REQUEST) has been added as a navigation data model element. When this query is made by an SCO, the LMS returns a value indicating whether the request is valid. The query result is *True* if the request is valid, *False* if it is not valid, and *Unknown* if the validity is not known.

`GetValue("adl.nav.request_valid.`<REQUEST>`")`
where <REQUEST> is *continue, previous* or *choice*.
Return value: *true, false* or *unknown*

To check whether a choice navigation request is valid, the identifier of the target activity must be specified as shown below.

`GetValue("adl.nav.request_valid.choice {target=`<STRING>`}")`
where <STRING> indicates the item identifier of the target activity.

In addition, an SCO can confirm the value that is currently set to the adl.nav.request element through the *GetValue* method. The value returned from a *GetValue* method call is the value currently stored at the data element on the LMS side. If there is no value set to the element, *_none_*, which is the initial value of *adl.nav.request*, is returned.

`GetValue("adl.nav.request",` <REQUEST>)
where <REQUEST> is one of the following: *continue, previous*, *choice, exit, exitAll, abandon*, or *abandonAll*.

4.3 Controlling LMS-Provided Navigation Devices

SCORM 2004 allows a content developer to specify whether the user interface devices of the LMS are to be hidden or shown. By including navigation request events in SCOs and specifying the user interface device control on the LMS, content developers can introduce their own design policies to their content user interface and content organization.

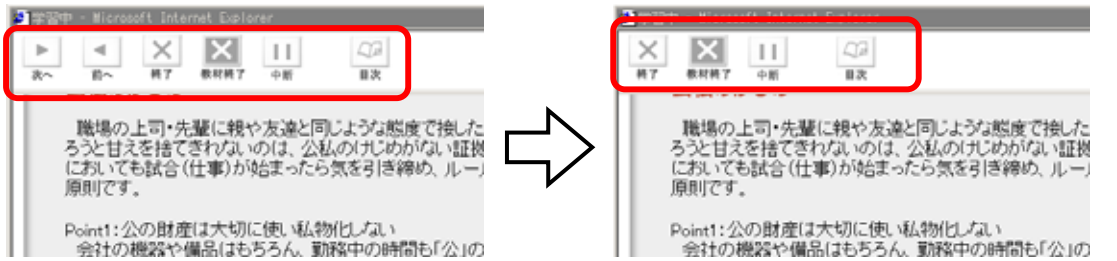
It is possible to specify whether to show or hide the LMS-provided user interface devices corresponding to the *continue*, *previous*, *exit* and *abandon* navigation requests by defining an appropriate token as the value of the *hideLMSUI* element for each activity in the manifest file (imsmanifest.xml). Table 4.2 shows a list of vocabulary tokens that can be used with the *hideLMSUI* element.

Table 4.2 Tokens for Controlling LMS-provided Navigation Devices

Token	Description
previous	Hide the *Previous* navigation device
continue	Hide the *Continue* navigation device
exit	Hide the *Exit* navigation device
abandon	Hide the *Abandon* navigation device

The example below specifies a situation where the *Continue* and *Previous* navigation user interface devices provided by the LMS will not be displayed at the run-time for Activity “item1”.

```
<organization>
   <item identifier=”item1” identifierref=”Resource1” isvisible=”true”>
          <adlnav:presentation>
                 <adlnav:navigationInterface>
                        <adlnav:hideLMSUI>continue</adlnav:hideLMSUI>
                        <adlnav:hideLMSUI>previous</adlnav:hideLMSUI>
                 </adlnav:navigationInterface>
          </adlnav:presentation>
   </item>
</organization>
```



Without hideLMSUI specified

With hideLMSUI specified

Figure 4.4 Displaying and Hiding LMS-provided Navigation Devices

5 Run-Time Environment (RTE)

The section describes the SCORM Run-Time Environment that lies between learning resources like SCOs and the LMS. It covers the changes made to the Run-Time Environment of SCORM 2004 from that of SCORM 1.2.

5.1 SCORM 2004 Run-Time Environment Overview

The SCORM Run-Time Environment (RTE) book describes a common launching content object mechanism of learning resources, a common communication mechanism between learning resources and the LMS, and a common data model for handling the tracking information for managing each learner's performance and progress with the learning resources. In a Run-Time Environment, an SCO that has been delivered with the use of an Application Programming Interface (API) instance communicates with the LMS (Figure 5.1).

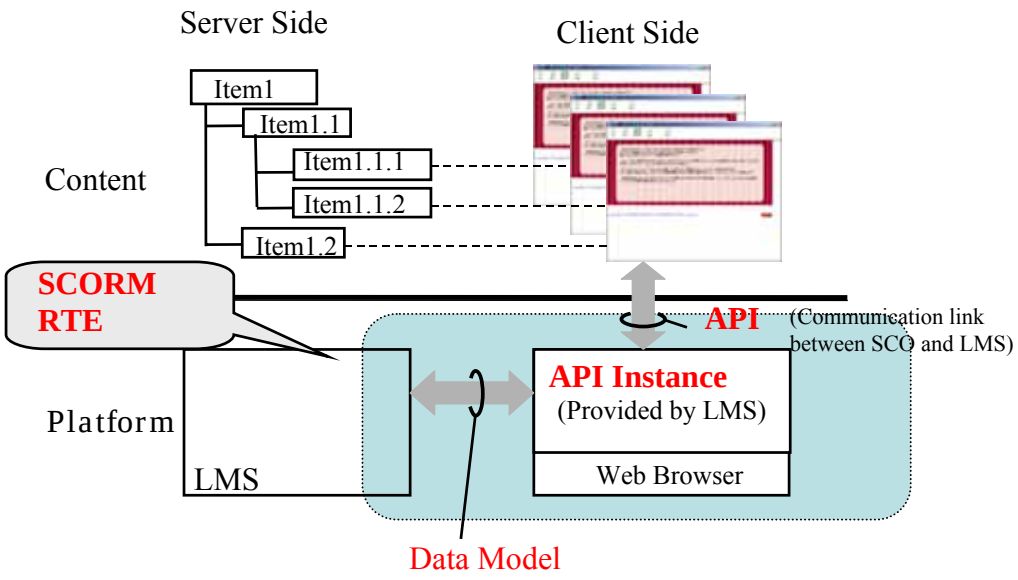


Figure 5.1 SCORM 2004 Run-Time Environment

The launching process defines common methods used by the LMS to launch Web-based learning resources. This mechanism specifies the methods and responsibilities for establishing a communication session between the LMS and a Content Object that has been delivered. The communication mechanism is standardized with a common API.

The API is a communication mechanism for sending status information regarding the Content Object (for example, regarding the initialization, termination, and error status) and is used for setting or getting data between the launched SCO and the LMS. The API Instance in Figure 5.1 indicates a software component that is provided by the LMS for a launched SCO to locate and use for communication with the LMS in an ECMAScript compatible language (e.g., JavaScript).

The data model is a standard set of data elements defined for storing and referencing the necessary information to be tracked such as the completion status of an SCO and the score from a quiz or test assessment. The LMS and SCOs are implemented in such a way that the LMS and the SCOs communicate with each other using the data model elements under the assumption that the other party knows the meanings and uses of these data model elements.

5.2 Launching Content Objects

The LMS is responsible for determining a learning activity to be attempted based on a navigation request, and then delivering an associated learning resource for the learner. When delivering a learning resource, the LMS launches the target resource using the URL that has been defined as the launch location for the learning resource. A launch method may be implemented either on the client side or the server side, but the learning resource whose launch location is defined using the HTTP protocol is displayed on the Web browser window of the client.

There are two types of learning resource that can be launched by the LMS: SCOs and assets.

5.2.1 Assets

An asset refers to a resource consisting of digital media such as text and images that can be loaded through a Web browser. An asset does not communicate with the LMS. An asset does not need to include a function called to the API that is provided by the LMS.

5.2.2 SCOs

The standard specifies that as a collection of one or more assets, a Sharable Content Object (SCO) should communicate with the LMS using the Run-Time Environment. An SCO is the minimum resource unit whose behavior can be recorded and managed by the LMS.

It is also specified in SCORM that only one SCO can be launched by the LMS at a given time and one SCO can be activated.

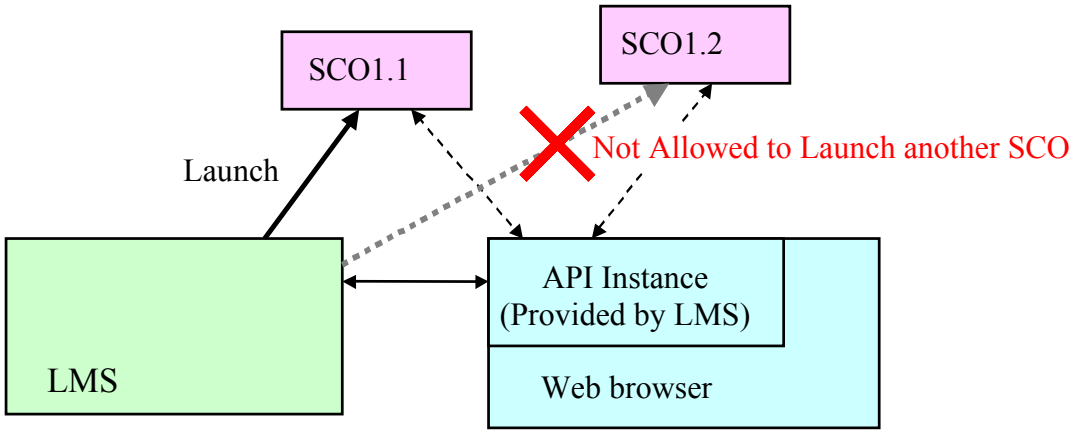


Figure 5.2 Launching an SCO

5.3 API

5.3.1 API Overview

Prior to its current version (SCORM 2004), SCORM was based on the AICC *CMI001 Guidelines for Interoperability*. The AICC has submitted its work to the IEEE Learning Technology Standards Committee (LTSC) as a candidate to become an international standard. SCORM has adopted the *IEEE 1484.11.2-2003 Standard for Learning Technology – ECMAScript Application Programming Interface for Content* as the standard for Runtime Services (RTS) communication. Adopting a common API made it possible for SCORM to meet its high-level requirements concerning interoperability and usability. The API provides a standardized way for SCOs to communicate with LMSs, and also enables LMS implementers to encapsulate their own specific implementation details from content developers. The LMS has to provide the API, API Implementation and API Instance that SCOs need to communicate.

5.3.2 API Instance Overview

An API Instance[3] is a piece of software that implements and exposes the necessary API methods, and it is provided to an SCO by the LMS as an interface through which the SCO can communicate with the LMS. Content developers must develop their contents (SCOs) in such a way that they can locate this API Instance provided by the LMS.

An important aspect of the API is that it allows an SCO to communicate with the LMS. Once an SCO has been launched, the SCO can store (*SetValue*) or refer to (*GetValue*) the data that the LMS is keeping for the SCO. Communication between the API Instance and an SCO is realized by invoking the methods of the API Instance.

The name of the API Instance is "API_1484_11"[4] in SCORM 2004.

[3] The API Instance was called the API adapter in SCORM versions prior to 1.2.
[4] The API Instance name has been changed from "API" to "API_1484_11" in SCORM 2004.

5.3.3 Using the API Instance

To establish a communication session with the LMS, a launched SCO must find the instance of the API implementation object that has been provided to it by the LMS. This means that the SCO must recursively search the parent windows and opener window for the API Instance. In this case, the LMS must make the API Instance available in the Document Object Model (DOM) context as an object called "*API_1481_11*".

5.3.3.1 LMS Responsibilities

The LMS must provide an API Instance under the following conditions:

- The LMS must make it possible for SCOs to access the API Instance in the DOM context with the name of "*API_1481_11*".
- The LMS must enable the SCO to access the API Instance using ECMAScript (JavaScript) code.
- The LMS must launch an SCO in a child window of the window where the API Instance has been loaded or in a child frame of the LMS window.

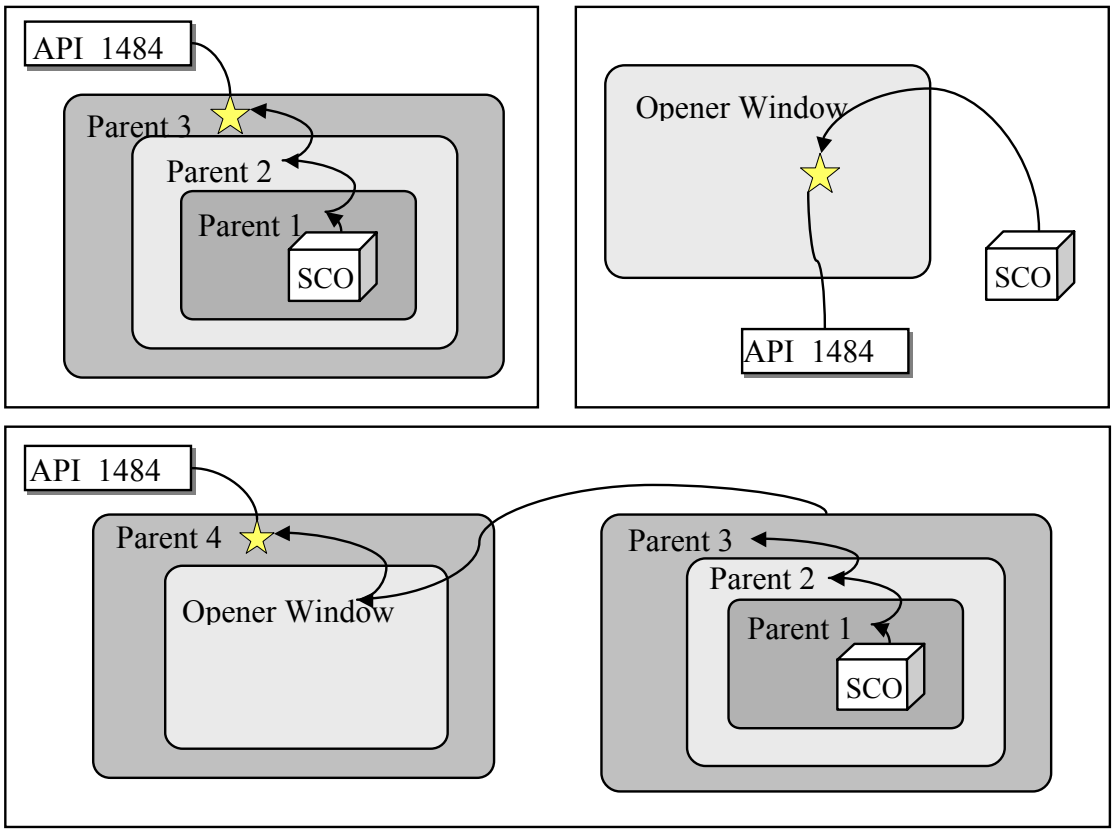


Figure 5.3 Permitted Location of an API Instance

5.3.3.2 SCO Responsibilities

SCOs must be developed in such a way that they can establish a communication session with the LMS by searching for the API Instance. For an SCO to find the API Instance located in a DOM window, it must search in the following sequence:

1. Search the chain of parents of the current window until the top window of the parent chain is reached.
2. Search the opener window (window.opener), which was the window opened by the SCO.
3. Search the chain of parents of the opener window, if any exist, until the top window of the parent chain is reached.

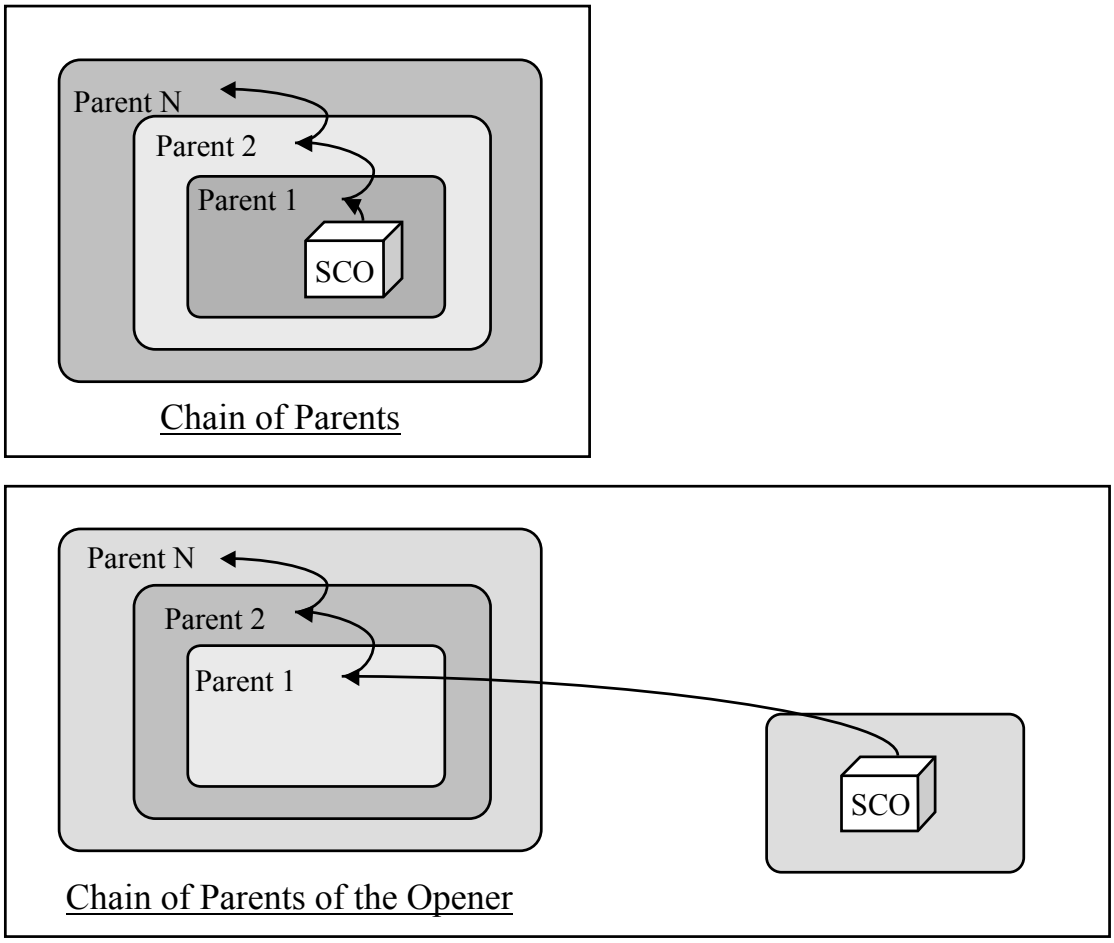


Figure 5.4 Finding the API Instance

The minimum amount of communication that an SCO must have with the LMS after locating the API Instance is the invocation of the *Initialize("")* and *Terminate("")* functions. The IEEE standard has provided a simple piece of ECMAScript that can be used to find the API Instance in a consistent manner. However, the standard does not require use of this ECMAScript code. Other means can be adopted.

Below is a sample code for searching recursively for the API Instance.

```
<html>
<head>
<script type=”text/javascript”>
<!--
   //----- Find the API Instance -----
   var API = null;
   function FindAPI(win) {
          if ((typeof(win.API_1484_11) != “undefined”) &&
                       (win.API_1484_11 != null)) {
                 return win.API_1484_11;
          } else if (win.location == top.location) {
                 return null;
          } else {
                 return FindAPI(win.parent);
          }
   }
   function MyInit() {
          // Find an API frame.
          if ((window.parent != null) && (window.parent != window)) {
                 API = FindAPI(window.parent);
          }
          // Find the API Instance in an opener window.
          if ((API == null) && (window.opener != null)) {
                 API = FindAPI(window.opener);
          }
          if (API != null) {
                 // Initialize
                 API.Initialize("");
          } else {
                 alert("Cannot find the API Instance。");
          }
   }
   function MyFin() {
          if (API != null) {
                 // Terminate the session
                 API.Terminate("");
          }
   }
//-->
</script>
</head>
<body onload=”MyInit();” onunoad=”MyFin()”>
   <h1>SCORM Sample Code</h1>
</body>
</html>
```

5.3.4 API Method Overview

The API methods are classified into three categories as shown below.

Table 5.1 Categories of API Methods

Category	Description	API methods
Session Methods	Session methods are used to mark the beginning and end of a communication session between an SCO and an LMS through the API Instance.	*Initialize* *Terminate*
Data-transfer Methods	Data-transfer methods are used to exchange data model element values between an SCO and an LMS through the API Instance.	*GetValue* *SetValue* *Commit*
Support Methods	Support methods are used for auxiliary communications (e.g., error handling) between an SCO and an LMS through the API Instance.	*GetLastError* *GetErrorString* *GetDiagnostic*

In SCORM 2004, the names of the API methods provided by the LMS have been changed. (As the LMS prefix has been dropped from the names, it has become easier to recognize the names.)

Table 5.2 Changes to API Method Names

SCORM 1.2	SCORM 2004
LMSInitialize	Initialize
LMSFinish	Terminate
LMSGetValue	GetValue
LMSSetValue	SetValue
LMSCommit	Commit
LMSGetLastError	GetLastError
LMSGetErrorString	GetErrorString
LMSGetErrorDiagnostic	GetErrorDiagnostic

The syntactic details of the API methods are shown in Table 5.3.

Table 5.3 List of API Methods

Session Methods	
Initialize	Syntax: *Initialize (*parameter*)* Description: This method is used to initialize a communication session. Parameter: (“”) an empty character string Return value: A character string indicating a Boolean value (true or false). ”true”: Indicates that the initialization on the LMS side was successful. ”false”: Indicates that the initialization on the LMS side was not successful. In this case, the API Instance sets an error code. Support methods are used to interpret the error data.
Terminate	Syntax: *Terminate (*parameter*)* Description: This method is used to terminate the communication session. The termination process should also transmit the data that the SCO has set with the API Instance but has not yet been stored in the LMS. Once this method has been executed, it is not possible to call any support methods. Parameter: (“”) an empty character string Return value: A character string indicating a Boolean value (true or false). ”true”: Indicates that the termination on the LMS side was successful. ”false”: Indicates that the termination on the LMS side was not successful. In this case, the API Instance sets an error code. Support methods are used to interpret the error data.

Data Transfer Methods	
GetValue	Syntax: *GetValue(*parameter*)* Description: This method is used to request information from the LMS. The SCO can request the following information from the LMS: • The values of the data model elements supported by the LMS • The version of the data model supported by the LMS • Whether specific data model elements are supported Parameter: The parameter indicates the identification of the target data model element. Return value: One of two types of character string: • A character string representing the value of the data model element indicated by the parameter • An empty character string (“”) when an error occurs. In this case, the API Instance sets an error code. Support methods are used to interpret the error data.
SetValue	Syntax: *GetValue(*parameter_1, parameter_2*)* Description: This method is used to set the value indicated by parameter_2 as the value of the data element indicated by parameter_1 at the LMS. The data may be instantly transmitted to the LMS or sent in a batch after being cached for a while, depending on the design as follows: • The values of the data model elements supported by the LMS • The version of the data model supported by the LMS • Whether specific data model elements are supported Parameter: parameter_1 indicates the name of the target data element parameter_2 indicates the value to be stored (a character string) Return value: A character string indicating a Boolean value (true or false). ”true”: Indicates that the data transfer to the LMS side was successful. ”false”: Indicates that the data transfer to the LMS side was not successful. In this case, the API Instance sets an error code. Support methods are used to interpret the error data.

Data Transfer Methods	
Commit	Syntax: *Commit(*parameter*)* Description: This method is used to commit stored data from the SCO to the LMS. If there is any data from the SCO that has been cached by the API Instance since the last call to *Initialize("")* or *Commit("")*, whichever occurred most recently. If the commit is successful, the LMS sets the error code to "0" (no error encountered) and returns "true". If the API Instance does not cache any data, this method is processed in the same way as above. Parameter: ("") an empty character string. Return value: A character string indicating a Boolean value (true or false). "true": Indicates that the commit process on the LMS side was successful. "false": Indicates that the commit process on the LMS side was not successful. In this case, the API Instance sets an error code. Support methods are used to interpret the error data.
Support Methods	
GetLastError	Syntax: *GetLastError()* Description: This method is used to request the error code for the latest error state of the API Instance. The API Instance does not alter the state of the current error upon a call of this method by the SCO, and simply returns the error code. Parameter: No parameter is specified. Return value: A character string indicating the error code for the current error state.
GetErrorString	Syntax: *GetErrorString(*parameter*)* Description: This method is used to request a textual description of the current error state. The API Instance should guarantee the support for the error code implemented at the API. The API Instance does not alter the state of the current error upon a call of this method by the SCO, and simply returns a character string which is the error description. Parameter: A character string indicating a target error code. Return value: A character string representing the error message corresponding to the error code indicated by the parameter. • The maximum length of a return value character string is 255 characters. • While a set of error codes is explicitly specified by SCORM, the description of each error code is specific to the LMS. • When the LMS cannot identify the error code, an empty character string ("") is returned.

Support Methods	
GetDiagnostic	Syntax: *GetDiagnostic(*parameter*)* Description: This method is provided for specific use of the LMS. It allows the LMS to define additional diagnostic information through the API Instance. Parameter: An implementer-specific value for diagnostics. The maximum length of the parameter value is 255 characters. An error code may be used as the parameter, but the parameter can also be used for other purposes. Return value: A character string representing the diagnostic information that is implemented by the LMS. The maximum length of a return value character string is 255 characters. Note: If the *GetDiagnostic( )* function is called with an empty character string ("") as the parameter, it is recommended that the function should return a character string representing diagnostic information about the last error encountered.

5.3.5 API Instance State Transitions

A conceptual model has been defined to specify the transitions of the API Instance during execution. The states of the API Instance indicate the transitions of the API Instance for specific events. These states are defined as

- Not Initialized
- Running
- Terminated

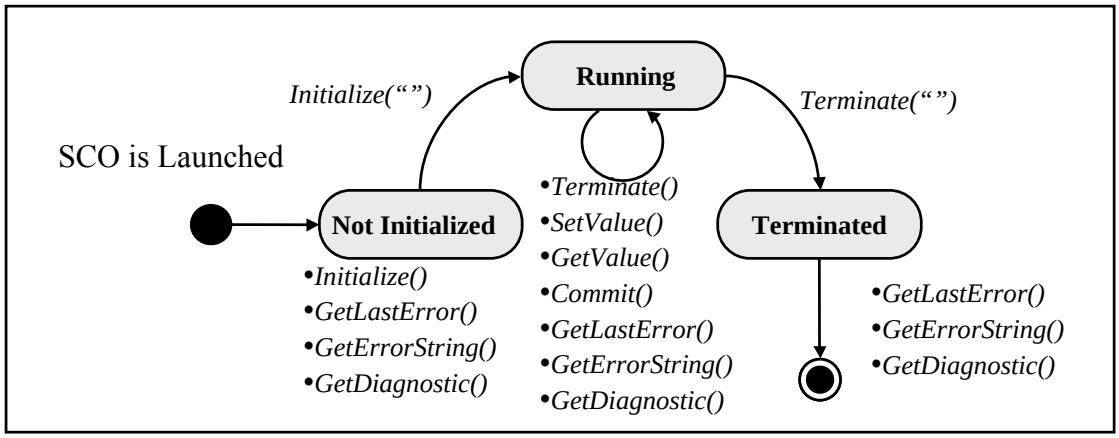


Figure 5.5 API Instance State Transitions and SCORM API

(1) Not Initialized

This is a state where no communication session has been established. It is the state before the SCO has successfully invoked the *Initialize* API method. The SCO is permitted to call the following set of API functions:

- *Initialize()*

- *GetLastError()*
- *GetErrorString()*
- *GetDiagnostic()*

(2) Running

This is a state in which the SCO exchanges data with the LMS after establishing a communication session. The SCO is permitted to call the following set of API methods:

- *Terminate()*
- *SetValue()*
- *GetValue()*
- *Commit()*
- *GetLastError()*
- *GetErrorString()*
- *GetDiagnostic()*

(3) Terminated

This is a state in which the SCO has successfully invoked the *Terminate* function. The SCO is permitted to call the following set of API methods:

- *GetLastError()*
- *GetErrorString()*
- *GetDiagnostic()*

5.3.6 API Error Code Overview

All error codes are character strings representing integers (0 – 655535). The IEEE standard has reserved codes between 0 and 999. Additional error codes may be defined by implementers using the remaining numbers from 1000 to 655535. The IEEE has defined the error code categories shown in Table 5.4.

Table 5.4 Categories of Error Codes

Error Code Category	Error Code Range	Description
No Error	0	The LMS returns this value when there is no error
General Errors	100 – 199	Errors that occur during the handling of API methods
Syntax Errors	200 – 299	Syntax errors found in the invoked API methods
RTS Errors	300 – 399	Errors associated with the implementation of the run-time system
Data Model Errors	400 – 499	Errors in the data sent to the LMS or the data received from the LMS
Implementation-defined Errors	1000 - 65535	For LMS implementers to use for their specific purposes

Table 5.5 shows the details of the API implementation errors.

Table 5.5 API Error Code Details

Code	Message	Description	API Methods
0	`No Error`	This code is returned when there is no error state.	All methods
101	`General Exception`	General errors occurred during the processing of API method requests.	*`Initialize()`*
102	`General Initialization Failure`	An error occurred during the initialization process of a communication session. The communication state remains "*Not Initialized*".	*`Initialize()`*
103	`Already Initialized`	After the communication was successfully established, the SCO attempted to initialize it again.	*`Initialize()`*
104	`Content Instance Terminated`	The SCO attempted to invoke the *Initialize* method after the communication session was terminated.	*`Initialize()`*
111	`General Termination Failure`	A general failure occurred when an attempt was made to terminate the session.	*`Terminate()`*
112	`Termination Before Initialization`	The SCO attempted to terminate a communication session before initializing the session.	*`Terminate()`*
113	`Termination After Termination`	The SCO attempted to terminate the communication session after the communication session was successfully terminated.	*`Terminate()`*
122	`Retrieve Data Before Initialization`	The SCO attempted to retrieve data before it successfully initialized a communication session.	*`GetValue()`*
123	`Retrieve Data After Termination`	The SCO attempted to retrieve data after termination of the corresponding communication session.	*`GetValue()`*
132	`Store Data Before Initialization`	The SCO invoked a *SetValue* method of the API Instance to store data before it successfully initialized a communication session.	*`SetValue()`*
133	`Store Data After Termination`	The SCO invoked a *SetValue* method of the API Instance to store data after termination of the corresponding communication session.	*`SetValue()`*

Code	Message	Description	API Methods
142	`Commit Before Initialization`	The SCO invoked a *Commit* method to save its data to a persistent store in the LMS before it successfully initialized a communication session.	*`Commit()`*
143	`Commit After Termination`	The SCO invoked a *Commit* method to save its data to a persistent store of the LMS after termination of its communication session.	*`Commit()`*
201	`General Argument Error`	An attempt was made to pass an invalid argument to the invoked API method.	*`Initialize()`* *`Terminate()`* *`Commit()`*
301	`General Get Failure`	A general failure occurred during the attempt to retrieve the requested data and there is no other error information available. The communication state remains "*Running*" in this situation.	*`GetValue()`*
351	`General Set Failure`	A general failure occurred during the attempt to set the requested data and there is no other error information available. The communication state remains "*Running*" in this situation.	*`SetValue()`*
391	`General Commit Failure`	A general failure occurred during the attempt to commit the data and there is no other error information available. The communication state remains "*Running*" in this situation.	*`Commit()`*
401	`Undefined Data Model Element`	The invoked API method contains a parameter that the API Instance cannot recognize. The communication state remains "*Running*" in this situation.	*`GetValue()`* *`SetValue()`*

Code	Message	Description	API Methods
402	`Unimplemented Data Model Element`	The invoked API method contains a parameter that has not been implemented by the LMS. The communication state remains "*Running*" in this situation. This should not occur when accessing SCORM data model elements, but may occur when accessing extension data model elements.	*`GetValue()`* *`SetValue()`*
403	`Data Model Element Value Not Initialized`	The SCO attempted to retrieve the value of a data model element that has not been initialized. Note that some data model elements are initialized by an SCO while others are initialized by the LMS.	*`GetValue()`*
404	`Data Model Element Is Read Only`	The SCO attempted to set a new value to a read-only data model element.	*`SetValue()`*
405	`Data Model Element Is Write Only`	The SCO attempted to retrieve the value of a write-only data model element.	*`GetValue()`*
406	`Data Model Element Type Mismatch`	The SCO attempted to store a value that was of an incorrect data type for the target data model element.	*`SetValue()`*
407	`Data Model Element Value Out Of Range`	The SCO attempted to store an invalid value that was outside the specified range for the target data model element.	*`SetValue()`*
408	`Data Model Dependency Not Established`	This error code is designed for handling a situation where an SCO attempts to read the value from, or write a value to, a data model element that has a dependence relationship defined with one or more other elements when the dependence has not yet been established.	*`GetValue()`* *`SetValue()`*

Table 5.6 compares the SCORM 1.2 and SCORM 2004 error codes in conjunction with the API Instance state transitions.

Table 5.6 Comparison of Error Codes between SCORM 1.2 and SCORM 2004

SCORM 1.2 Error Code	SCORM 2004 Error Code
0 – No error	0 - No error
101 – General Exception	101 - General Exception
	102 - General Initialization Failure
	103 - Already Initialized
	104 - Content Instance Terminated
	111 - General Termination Failure
	112 - Termination Before Initialization
	113- Termination After Termination
	122 - Retrieve Data Before Initialization
	123 - Retrieve Data After Termination
	132 - Store Data Before Initialization
	133 - Store Data After Termination
	142 - Commit Before Initialization
	143 - Commit After Termination
201 - Invalid argument error	201 - General Argument Error
202 - Element cannot have children 203 - Element not an array. Cannot have count	301 - General Get Failure
	351 - General Set Failure
	391 - General Commit Failure
401 - Not implemented error	401 - Undefined Data Model Element
401 - Not implemented error	402 - Unimplemented Data Model Element
301 - Not initialized	403 - Data Model Element Value Not Initialized
403 - Element is read only	404 - Data Model Element Is Read Only
404 - Element is write only 402 - Invalid set value, element is a keyword	405 - Data Model Element Is Write Only
405 - Incorrect Data Type	406 - Data Model Element Type Mismatch
	407 - Data Model Element Value Out Of Range
	408 - Data Model Dependency Not Established

5.3.7 API Error Handler Implementation Example

An SCO should contain an API implementation error handler that can check and interpret the error code set by the API Instance at each state transition.

An example of code that can be used to check error states is shown below.

```
<html>
<head>
<script type=”text/javascript”>
<!--
   var API = null;
   function FindAPI(win) {
        (Omitted)
   }
   function MyInit() {
         (Omitted)
   }
   function MyFin() {
         (Omitted)
   }
//----- Check errors-----
    function CheckError() {
        var errMsg = "";
        if (API != null) {
         if (parseInt(API.GetLastError()) > 0) {
                errMsg = API.GetErrorString() + ":" +
                         API.GetDiagnostic();
                alert(errMsg);
                }
         }
    }
//-->
</script>
</head>
<body onload=”MyInit();” onunoad=”MyFin()”>
   <h1>SCORM Sample Code</h1>
</body>
</html>
```

5.4 Data Model

5.4.1 Data Model Overview

The SCORM Run-Time Environment Data Model specification is based on the *IEEE P1484.11.1 Draft Standard for Learning Technology – Data Model for Content Object Communication*. This standard defines a set of data model elements that can be used for communication from SCOs to the LMS. The data model includes, among other things,

- Information about the learner
- Interactions between an SCO and the LMS
- Objectives
- Success status and completion status.

The data model elements are defined in such a way that they can be used for a variety of content purposes.

The main uses of the Run-Time Environment data are

- To track the learner's progress and status
- To support sequencing decisions
- To report on the learner's overall interactions with the SCO.

The SCORM Version 1.2 Run-Time Environment Data Model was based on the *AICC CMI001 Guideline for Interoperability*. Since the release of SCORM Version 1.2, AICC has submitted CMI001 to the IEEE for standardization. SCORM 2004 has introduced changes to the data model in accordance with *the IEEE 1484.11.1 Draft Standard*. The major changes can be summarized as follows:

- All data model elements have become mandatory for the LMS to implement.
- Changes to the data model
    - Removal of *cmi.core* and *cmi.student_data*
    - Addition of *score.scaled*
    - Addition of sequencing data model elements corresponding to *objectives*
- Detailed specification of interactions
- Adoption of Unicode (ISO-10646-1) for multi-language implementation support, including multi-byte code.

5.4.2 Data Model Basics

5.4.2.1 Data Model Elements

To differentiate them from the other data model elements, the names of all Run-Time Environment Data Model elements start with "*cmi*". This indicates to the LMS that these data model elements are part of the IEEE P148411.1 standard. It is anticipated that when a different data model is to be developed, its data elements will be named with a different prefix (for example, *adl.elementName* instead of *cmi.elementName*).

It is mandatory for the LMS to implement all these data model elements and to guarantee their behaviors.

Content developers can choose to use any or all of the data model elements in SCOs.

The names of all the data model elements must be bound to ECMAScript character strings with dot notation (for example, *cim.success_status*).

5.4.2.2 Data Model Effects on Sequencing

Using the Run-Time Data Model elements, each SCO reports the results of interactions between the learner and the SCO to the LMS during a session. The LMS uses the information reported from an SCO when it makes a sequencing decision as to which activity is to be delivered next. For example, when an SCO reports its attempt status as completed (by the learner) to the LMS using the data element "*cmi.completion_status*" (as tracking information), the LMS regards the activity associated with the SCO as completed and selects another activity for delivery. Some data model elements of the RTE are related to the tracking information for each activity, and thus affect the sequencing process. Refer to Section 7.4 for the relationships between tracking information and the RTE Data Model.

5.4.2.3 Handling Collections

Some data model elements are collections of related elements defined for specific requirements. Such collection data is referred to as a record of data in the SCORM RTE book. Each record of data is collected as an entity in an array. The record of data is accessed using an index value representing the record of data's position in the array. All arrays are implemented with a starting index of 0 (zero-based arrays).

The following data model elements are defined as collections of data records:

- Comments from learner (*cmi.comments_from_learner*)
- Comments from LMS (*cmi.comments_from_lms*)
- Objectives (*cmi.objectives*)
- Interactions (*cmi.interactions*)

These data model elements are to ensure that SCOs can track multiple comments, objectives and/or interactions. The Objectives and Interactions data model elements contain an identifier data model element that indicates a unique identifier for each of the SCO's Objectives and Interactions.

The data model elements in a collection are referred to using a dot-number notation (represented as *n* in the following.)

cmi.objective.n.completion_status

For example, the value of the data model element representing the completion status of the first objective in an SCO is described as "*cmi.objective.0.completion_status*", and that of the fourth objective is described as "*cmi.objective.3.completion_status*". The *_count* keyword data model element is used to determine the current number of data model elements in the collection. For example, to determine the number of

objectives currently stored for the SCO, the following API method call would be used:

```
var numOfObjectives = GetValue("cmi.objectives._count");
```

5.4.2.4 Smallest Permitted Maximum (SPM)

In SCORM 2004, the smallest permitted maximums (SPMs) are defined for the data model elements in two cases. The SPM is defined as the length of a character string value or the number of entries (data model elements) in collections; that is, the SPM is defined as the smallest permitted length of a character string that any implementation must accept and process or the smallest permitted number of entries in a collection. An implementation may elect to support the storage of more than the SPM. If an implementation only supports the SPM and truncates a character string, it is necessary for content developers to be aware of the SPM and what may happen if it is exceeded.

5.4.2.5 Keyword Data Model Elements

SCORM defines a set of data model elements for getting the data managed by the LMS and the status data of some data model elements. These data model elements are called keyword data model elements. The keyword data model elements can only be applied to certain data model elements and are implemented as read-only data model elements.

- *_version*: The *_version* keyword data model element is used to retrieve the version of the data model supported by the LMS.
- *_count*: The *_count* keyword data model element is used to retrieve the number of data model elements contained in a collection.
- *_children*: The *_children* keyword data model element is used to retrieve the entire set of child data model elements included in a parent data model element supported by the LMS. The LMS must implement the return value for this *_children* request as a list of character strings delimited by a comma, with each string representing a child data model element. This data model element can only be applied to a data model element that has its child data model elements.

5.4.2.6 Reserved Delimiters

A special reserved delimiter must be used to represent

- The language type for a particular character string (Data type: *localized_string_type*)
- The indication as to whether the order matters in the learner's responses to an interaction
- The indication as to whether the case matters in the learner's response to an interaction
- A set of values in a list or pairs of values.

For each of the above cases, a default value is provided where applicable. This default value is used if the special reserved delimiter is not specified. In any case, the reserved delimiters should not be counted toward the value of the SPM.

Table 5.7 Reserved Delimiters

Reserved Delimiter Syntax	Default Value	Example
`{lang=<language_type>}`	`{lang=en}`	`{lang=en}`
`{case_matters=<boolean>}`	`{case_matters=false}`	`{case_matters=true}` `{case_matters=false}`
`{order_matters=<boolean>}`	`{order_matters=true}`	`{order_matters=true}` `{order_matters=false}`
`[.]`	Not applicable. A value must be provided.	Used to separate a pair of values that are related for an interaction: `1[.]a`
`[,]`	Not applicable. A value must be provided.	Used to separate a set of values for an interaction's collection: `1[.]a[,]2[.]c[,]3[.]b`
`[:]`	Not applicable. A value must be provided.	Used to represent a separator between a range of numeric values: `1[:]100` – a range where the numeric value is between 1 and 100 (inclusive)

5.4.2.7 Data Types

Each of the data type elements has a designated data type. The values of a data model element must satisfy the data type requirements of the element. Below is a description of the specific requirements for each data model element.

(1) characterstring

A string of characters defined in ISO 10646, which is equivalent to the Unicode standard.

(2) localized string type

A character string that contains a character string indicating the language of the *characterstring*. SCORM uses a reserved delimiter to represent the language for a character string: `{lang=<language_type>}`. Whether to specify this localized string type is optional. If it is not specified, the default language is `en`, which means English as in `{lang=en}`. The syntax for specifying this data type element is as follows:

"`{lang=`<*language_type*>`}`<*actual_character_string*>"

Example: `{lang=ja}`鈴木 一郎

(3) language type

The data type used to represent the language. The format of this data type element is a character string consisting of a language code (langcode) followed by zero or more hyphen-prefixed subcodes (subcode).

`language_type ::=`*`langcode`* `[“-“`*`subcode`*`]*`

Example: `ja en-GB`

(4) long identifier type

This data type element represents a label or identifier. This label or identifier must be unique within the context of an SCO. It must conform to the syntax defined for the universal resource identifier (URI). SCORM recommends that the URI be a globally unique identifier in the form of a uniform resource name (URN). The values of this long identifier type element should be implemented with an SPM of 4000 characters.

`<URN>::=”urn.”<NID>”.”<NSS>`

where <NID> is a name space identifier and <NSS> is a name space string.

Example: `urn:ADL:interaction-id-0001`

(5) short identifier type

This label or identifier must be unique within the context of an SCO. This data type element must conform to the syntax defined for the URI. It is not assumed that the values of this data type element are globally unique identifiers. The values should be implemented with an SPM of 250 characters.

(6) integer

This data type element is a member of a set of positive whole numbers (e.g., 1, 2, 3), negative whole numbers (e.g., -1, -2, -3) and zero (0).

(7) state

Some of the data model elements values have a defined set of states. This is defined by a statement like the following:

`Example: state (browse,normal,review)`

(8) real(10,7)

This data type element refers to a real number with seven significant digit precision.

(9) time(second, 10, 0)

This time data type has a required precision of 1 second and an optional precision of 0.01 second.

Example: `2003-07-25T03:00:00`

(10) timeinterval(second, 10,2)

The value for this data type element represents a period of elapsed time with a precision of 0.01 second.

Example: `P1Y3M2DT3H`

which means 1 year, 3 months, 2 days and 3 hours.

5.4.2.8 SCORM Run-Time Environment Data Model Extension

The SCORM Run-Time Environment Data Model is expected to be implemented without extension. If an LMS receives an API request with an undefined data model element, the LMS should handle it as an error.

5.4.3 SCORM Run-Time Environment Data Model

5.4.3.1 Data Model Overview

The SCORM Run-Time Environment Data Model contains a set of data that can be tracked by an SCO on the LMS during the run-time of the SCO. These data model elements are used to track items such as status, scores, interactions, and objectives. Some data model elements are used to exchange data between the SCO and the LMS, while others may be used to affect the sequencing process for other SCOs that are associated within the activity tree. Table 5.8 summarizes the data model elements.

Table 5.8 List of SCORM Run-Time Environment Data Model Elements

No	Data Model Element	Description
1	*cmi.comments_from_learner*	Contains text from the learner
2	*cmi.comments_from_lms*	Contains comments and annotations to be provided for the learner
3	*cmi.completion_status*	Indicates whether the learner has completed the SCO
4	*cmi.completion_threshold*	Indicates a value against which the learner's measure of progress is to be compared to determine whether the learner has completed the SCO
5	*cmi.credit*	Indicates whether the learner's performance with the SCO is to be credited
6	*cmi.entry*	Contains information indicating whether the learner has accessed the SCO before
7	*cmi.exit*	Contains information as to why and how the learner exited from the SCO
8	*cmi.interactions*	Defines information concerning an interaction for the purpose of measurement or assessment
9	*cmi.launch_data*	Provides data specific to an SCO that the SCO can use for initialization
10	*cmi.learner_id*	Identifies the learner for whom the SCO instance was launched
11	*cmi.learner_name*	Represents the name of the learner
12	*cmi.learner_preference*	Specifies learner preferences associated with the learner's use of the SCO
13	*cmi.location*	Represents a location in the SCO. Its value and meaning are determined by the SCO.

No	Data Model Element	Description
14	*cmi.max_time_allowed*	Indicates the accumulated amount of time that the learner is allowed to use for an SCO in the learner attempt
15	*cmi.mode*	Identifies the modes in which the SCO may be presented to the learner
16	*cmi.objectives*	Specifies learning objectives associated with an SCO
17	*cmi.progress_measure*	Identifies a measure of the progress the learner has made toward completing the SCO
18	*cmi.scaled_passing_score*	Identifies the scaled passing score for an SCO
19	*cmi.score*	Identifies the learner's score for the SCO
20	*cmi.session_time*	Identifies the amount of time that the learner has spent in the current session for the SCO
21	*cmi.success_status*	Indicates whether the learner has mastered the SCO
22	*cmi.suspend_data*	Provides additional space for storing and retrieving information relating to the suspension of an SCO
23	*cmi.time_limit_action*	Indicates what the SCO should do when the maximum time allowed is exceeded
24	*cmi.total_time*	Provides the sum of all of the learner's session times accumulated in the learner's current attempt

5.4.3.2 Details of the Data Model Elements

Table 5.9 SCORM RTE Data Model Elements – Detail

No	Data Model Element	Data Type	Value Space	SCO	Remarks
0.1	*cmi._version*	characterstring	ISO-10646-1	R	Delimited by a period "1.0"
1.	*cmi.comments_from_learner*	collection SPM: 250		—	
1.0.1	*cmi.comments_from_learner._children*	characterstring	ISO-10646-1	R	
1.0.2	*cmi.comments_from_learner._count*	integer	Non-negative integer	R	
1.1	*cmi.comments_from_learner.n.comment*	localized_ string_type SPM: 4000	Localized string type (ISO-10646-1)	R/W	Not initialized
1.2	*cmi.comments_from_learner.n.location*	characterstring SPM: 250	ISO-10646-1	R/W	
1.3	*cmi.comments_from_learner.n.timestamp*	time (second,10,0)		R/W	
2.	*cmi.comments_from_lms*	collection SPM: 100		—	
2.0.1	*cmi.comments_from_lms._children*	characterstring	ISO-10646-1	R	
2.0.2	*cmi.comments_from_lms._count*	integer	Non-negative integer	R	
2.1	*cmi.comments_from_lms.n.comment*	localized_ string_type SPM: 4000	Localized string type (ISO-10646-1)	R	
2.2	*cmi.comments_from_lms.n.location*	characterstring SPM: 250	ISO-10646-1	R	
2.3	*cmi.comments_from_lms.n.timestamp*	time (second,10,0)		R	
3.	*cmi.completion_status*	state	*"complete" "incomplete" "not_attempted" "unknown"*	R/W	Default: "unknown" It is assumed that an SCO is to set the value of this element, and that it affects sequencing.
4.	*cmi.completion_threshold*	real(10,7) range (0..1)		R	LMS determines the completion status by comparing the 「17. cmi.progress _measure」 against the value. The result of this has a higher priority than 「3. cmi .completion _status」 which is set by an SCO. This is initialized with a value defined for <adlcp:completi onThreshold> in the manifest file.

* Notes on the SCO column: R: Read-only, W: Write-only, R/W: Read and Write

No	Data Model Element	Data Type	Value Space	SCO	Remarks
5.	*cmi.credit*	state	*"credit"* *"no_credit"*	R	Default: "credit"
6.	*cmi.entry*	state	*"ab_initio"* *"resume"* ""(Empty string)	R	
7.	*cmi.exit*	state	*"time-out"* *"suspend"* *"logout"* *"normal"* ""(Empty string)	W	
8.	*cmi.interactions*	collection SPM: 250		—	
8.0.1	*cmi.interactions._children*	characterstring	ISO-10646-1	R	
8.0.2	*cmi.interactions._count*	integer	Non-negative integer	R	
8.1	*cmi.interactions.n.id*	long_identifier_type SPM: 4000	URI (RFC 2396)-compliant character string URN (RFC 2141) Recommended	R/W	Must be unique within the context of an SCO.
8.2	*cmi.interactions.n.type*	state	*"true-false"* *"choice"* *"fill-in"* *"long-fill-in"* *"likert"* *"matching"* *"performance"* *"sequencing"* *"numeric"* *"other"*	R/W	The correct_response and learner_response elements are dependent on this data model element, and this must be set before these two dependent elements are used.
8.3	*cmi.interactions.n.objectives*	collection SPM: 10		—	
8.3.0.1	*cmi.interactions.n.objectives._count*	integer	Non-negative integer 0	R	
8.3.1	*cmi.interactions.n.objectives.n.id*	long_identifier_type) SPM: 4000	URI (RFC 2396)-compliant character string URN (RFC 2141) Recommended	R/W	
8.4	*cmi.interactions.n.timestamp*	Time (second,10,0)		R/W	
8.5	*cmi.interactions.n.correct_responses*	collection SPM: 10		—	
8.5.0.1	*cmi.interactions.n.correct_responses._count*	integer	Non-negative integer	R	

* Notes on the SCO column: R: Read-only, W: Write-only, R/W: Read and Write

No	Data Model Element	Data Type	Value Space	SCO	Remarks
8.5.1	*cmi.interactions.n.correct_responses.n. pattern*	Dependent on 8.2 cmi. interactions.n. type		R/W	
8.6	*cmi.interactions.n.weighting*	real (10,7)	Real with 7 significant decimal digits	R/W	
8.7	*cmi.interactions.n.learner_ response*	Dependent on 8.2 cmi. interactions.n. type		R/W	
8.8	*cmi.interactions.n.result*	state	*"correct" "incorrect" "unanticipated" "neutral" real(10,7)*	R/W	
8.9	*cmi.interactions.n.latency*	timeinterval (second,10,2)		R/W	
8.10	*cmi.interactions.n.description*	Localized _string_type SPM: 250	Localized string type	R/W	
9.	*cmi.launch_data*	characterstring SPM: 4000	ISO-10646-1	R	Initialized by the with a value defined for <adlcp:dataFrom LMS> in the manifest file.
10.	*cmi.learner_id*	long_identifier _type SPM: 4000	URI (RFC 2396)-compliant character string URN (RFC 2141) Recommended	R	Provided by the LMS
11.	*cmi.learner_name*	localized _string_type SPM: 250	Localized string type	R	Provided by the LMS
12..	*cmi.learner_preference*			-	
12.0.1	*cmi.learner_preference._children*	characterstring	ISO-10646-1	R	
12.1	*cmi.learner_preference.audio_level*	real(10,7), range (0..*)	A real number with 7 significant decimal digits	R/W	
12.2	*cmi.learner_preference.language*	language_type SPM: 250	ISO-646	R/W	
12.3	*cmi.learner_preference.delivery _speed*	real(10,7), range (0..*)	A real number with 7 significant decimal digits	R/W	
12.4	*cmi.learner_preference.audio _captioning*	state	*"-1" "0" "1"*	R/W	Corresponding tokens: *"off" "no_change" "on"*

* Notes on the SCO column: R: Read-only, W: Write-only, R/W: Read and Write

No	Data Model Element	Data Type	Value Space	SCO	Remarks
13.	*cmi.location*	characterstring SPM: 1000	ISO-10646-1	R/W	The initial value is “”（Empty string） An LMS should not update or interpret this value. This can be used to save the SCO’s exit point.
14.	*cmi.max_time_allowed*	timeinterval (second,10,2) With resolution to 0.01 second.		R	Initialized with a value defined for <imsss:attempt AbsoluteDuration Limit> in the manifest file.
15.	*cmi.mode*	state	*“browse”* *“normal”* *“review”*	R	Default: “normal” Related with：「5. cmi.credit」
16.	*cmi.objectives*	collection SPM: 100		—	
16.0.1	*cmi.objectives._children*	characterstring	ISO-10646-1	R	
16.0.2	*cmi.objectives._count*	integer	Non-negative integer	R	
16.1	*cmi.objectives.n.id*	long_identifier_type SPM: 4000	URI (RFC 2396)-compliant character string URN (RFC 2141) Recommended	R/W	Must be unique at least within the context of an SCO. Initialized with the value defined as the identifier for <imsss:objectives> in the manifest file.
16.2	*cmi.objectives.n.score*			—	
16.2.0.1	*cmi.objectives.n.score._children*	characterstring	ISO-10646-1	R	
16.2.1	*cmi.objectives.n.score.scaled*	real (10,7) range (-1..1)	Real number with 7 significant decimal digits within the range between -1.0 and 1.0	R/W	The data model element affects the objective measure for the activity associated with the SCO.
16.2.2	*cmi.objectives.n.score.raw*	real (10,7)	Real number with 7 significant decimal digits	R/W	

* Notes on the SCO column: R: Read-only, W: Write-only, R/W: Read and Write

No	Data Model Element	Data Type	Value Space	SCO	Remarks
16.2.3	*cmi.objectives.n.score.min*	real (10,7)	Real number with 7 significant decimal digits	R/W	
16.2.4	*cmi.objectives.n.score.max*	real (10,7)	Real number with 7 significant decimal digits	R/W	
16.2.5	*cmi.objectives.n.success_status*	state	*"passed"* *"failed"* *"unknown"*	R/W	The data model element affects the Objective Progress Status for the activity associated with the SCO.
16.2.6	*cmi.objectives.n.completion_status*	state	*"completed"* *"incomplete"* *"not_attempted"* *"unknown"*	R/W	
16.2.7	*cmi.objectives.n.progress_measure*	real (10,7) range (0..1)	Real number with 7 significant decimal digits within the range between 0 and 1.0.	R/W	
16.2.8	*cmi.objectives.n.description*	Localized _string_type SPM: 250	Localized string type (ISO-10646-1)	R/W	
17.	*cmi.progress_measure*	real (10,7) range (0..1)	Real number with 7 significant decimal digits within the range between 0 and 1.0	R/W	Mapped with the value of 「3. cmi.completion _status」 0 ⇒ "not attempted" 1⇒ " completed" 0 > value < 1 ⇒ " incomplete" Note) This applies when there is no defined threshold value.
18.	*cmi.scaled_passing_score*	real (10,7) range (-1..1)	Real number with 7 significant decimal digits within the range between -1.0 and 1.0	R	Initialized with the value defined for <imsss:minNormal-izedMeasure> in the manifest file.
19.	*cmi.score*			-	Mainly used by SCOs
19.0.1	*cmi.score._children*	characterstring	ISO-10646-1	R	

* Notes on the SCO column: R: Read-only, W: Write-only, R/W: Read and Write

No	Data Model Element	Data Type	Value Space	SCO	Remarks
19.1	*cmi.score.scaled*	real (10,7) range (-1..1)	Real number with 7 significant decimal digits within the range between -1.0 and 1.0	R/W	The value of this data model element must be synchronized with the initial value of the Objective Measure Status for the SCO.
19.2	*cmi.score.raw*	real (10,7)	Real number with 7 significant decimal digits	R/W	
19.3	*cmi.score.max*	real (10,7)	Real number with 7 significant decimal digits	R/W	
19.4	*cmi.score.min*	real (10,7)	Real number with 7 significant decimal digits	R/W	
20.	*cmi.session_time*	timeinterval (second,10,2) With resolution to 0.01 second		W	
21.	*cmi.success_status*	state	*"passed"* *"failed"* *"unknown"*	R/W	Initialized by the SCO The LMS cannot directly change this data model element, but it can indirectly change the value by setting a value to 「18. cmi.scaled _passing_score 」 （In this case, the LMS can override the value of the data model element reported by the SCO.） This data model element must be synchronized with the initial Objective Measure Status value for the SCO.

* Notes on the SCO column: R: Read-only, W: Write-only, R/W: Read and Write

No	Data Model Element	Data Type	Value Space	SCO	Remarks
22.	*cmi.suspend_data*	characterstring SPM: 4000	ISO-10646-1	R/W	The LMS should not attempt to interpret or change this data, which is meant to be used by the SCO for current or subsequent sessions. Related to：「13. cmi.location」
23.	*cmi.time_limit_action*	state	*"exit,message"* *"continue, message"* *"exit,no message"* *"continue, no message*"	R	Initialized with the value defined for <adlcp:timeLimitAction> in the manifest file. Default: "continue,no message"
24.	*cmi.total_time*	timeinterval (second,10,2) With resolution to 0.01 second		R	The LMS cannot determine the latest value for this data element until the SCO sets session time values to the cmi.session_time element.

* Notes on the SCO column: R: Read-only, W: Write-only, R/W: Read and Write

5.4.4 Data Model Implementation Example

The SCORM Run-Time Data Model is a collection of data model elements that are used between the LMS and SCOs. Both the LMS and SCOs communicate under the assumption that each party knows about these data model elements. The major communication behaviors include:

- Initialization
- Read data
- Write data
- Save / Store

5.4.4.1 Read-only Examples

(1) Description Examples

Example 1) Learner's ID

```
id = GetValue("cmi.learner_id");
```

Example 2) SCO's launch data

```
Lpm = getValue("cmi.launch_data");
```

Example 3) Maximum time allowed with the SCO

```
Time_limit = getValue("cmi_max_time_allowed");
```

(2) Behavior

- The LMS performs initialization using the session data and the information regarding the associated activity tree.
- The SCO utilizes this information after retrieving it from the Run-Time Environment.
- The SCO cannot update the values of these data elements.

(3) Description

- The initial value for Example 1) above is provided from the session information managed by the LMS.
- The initial values for Examples 2) and 3) are the values defined in the manifest file (imsmanifest.xml) as follows:

Example 2) Initialized with the value defined for <*adlcp:dataFromLMS*>, which is the launch data for the SCO.

Example 3) Initialized with the value defined for <*imsss:attemptAbsoluteDurationLimit*>, which indicates the learning time allowed for the learner on the SCO.

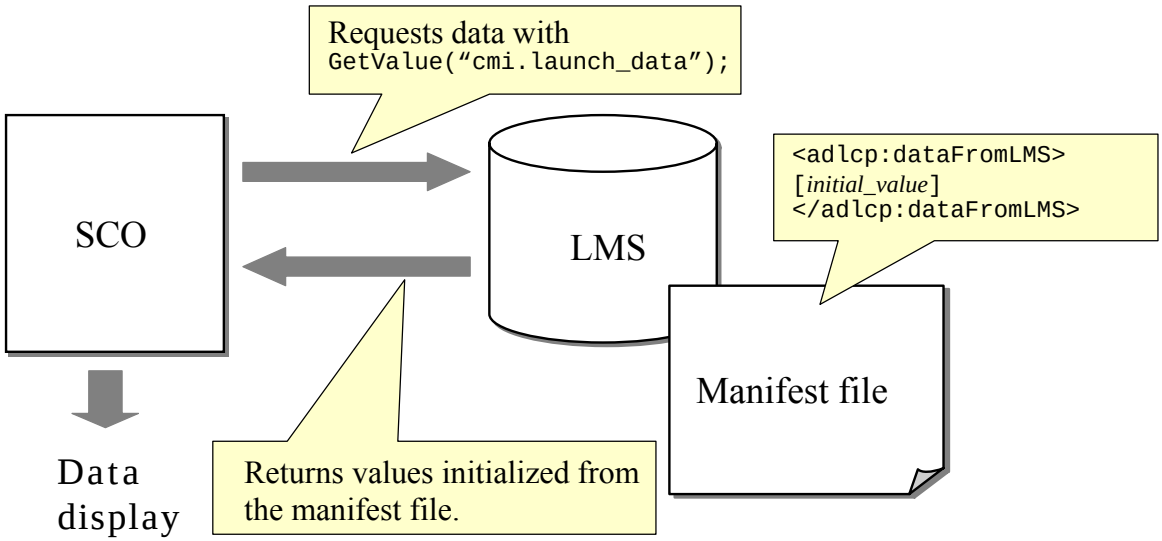


Figure 5.6 Getting Data Model Element Values

5.4.4.2 Write-only Examples

(1) Description Example

Example 4) Session time

```
SetValue("cmi.session_time", "05:15:00");
```

(2) Behavior

- The LMS does not initialize.
- The SCO sets a value to the specified data model element.

- The LMS processes, saves and stores the information.

(3) Description

- The data model elements are used mainly for storing the learner's performance and progress data with an SCO.

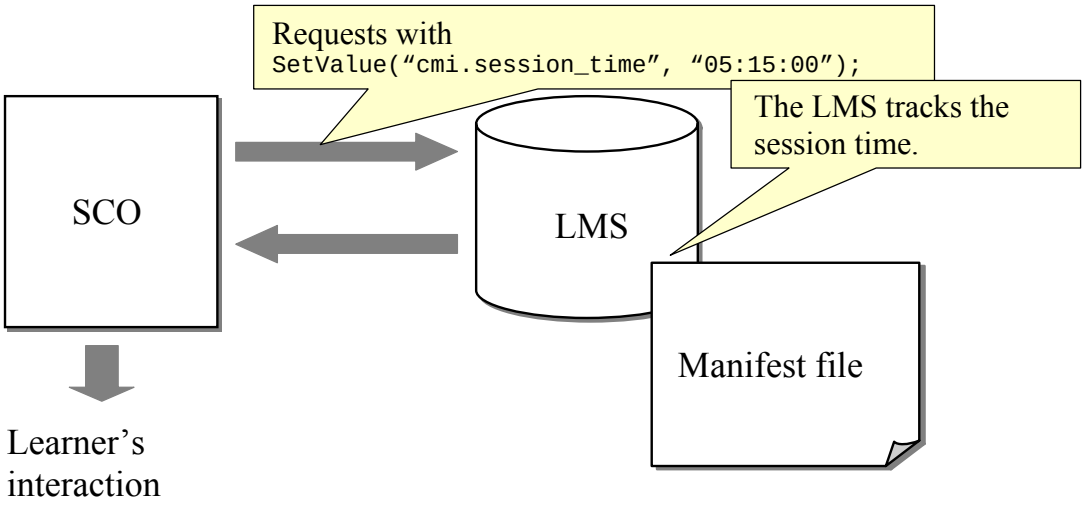


Figure 5.7 Setting Data Model Element Values

5.4.4.3 Read and Write Examples

(1) Description Example

Example 5) Reading the learner's success status.

```
Loc = GetValue("cmi.success_status");
```

Example 6) Writing the learner's success status data.

```
SetValue("cmi.success_status", "passed");
```

(2) Behavior

- The LMS initializes the associated data model elements as "*unknown*".
- The SCO uses the data after retrieving it and then updates it.
- The LMS processes, saves and stores the information.
- The SCO may use the data again after retrieving the updated data.

(3) Description

- The data model elements used for reading and writing are those used to get state changes during a session such as progress status, success status and performance measures.
- The SCO is supposed to initialize these data model elements.
- As state values and numerical values are used, errors are caused if the values are not within the defined set of keywords or within the allowed range.

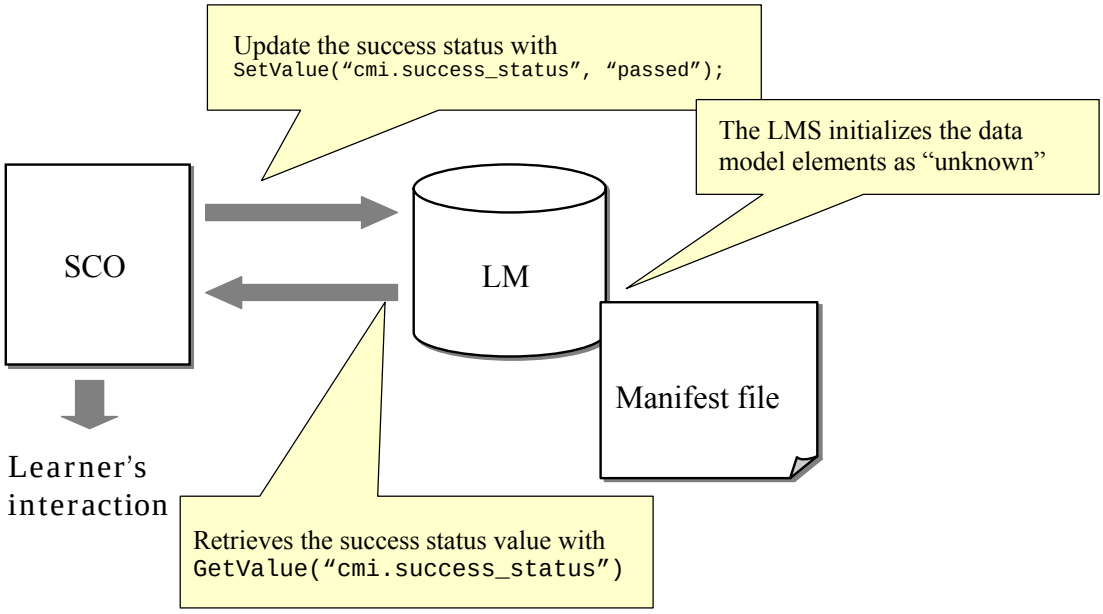


Figure 5.8 Getting and Setting Data Model Element Values

6 Sequencing Implementation

This section describes sequencing behaviors and their implementation methods. The sequencing behavior model is implemented by a set of procedures called sequencing processes. A request is passed between the sequencing processes, and a sequencing process is described by pseudo-code. An overview of the processing represented in this pseudo-code is given in this section.

6.1 Sequencing Process

As shown in Figure 3.1, the sequencing behavior is implemented through a set of procedures called sequencing processes. Although the SCORM SN book uses the term "behavior", this document has been standardized with the term "process". Figure 6.1 shows the relationship between these processes.

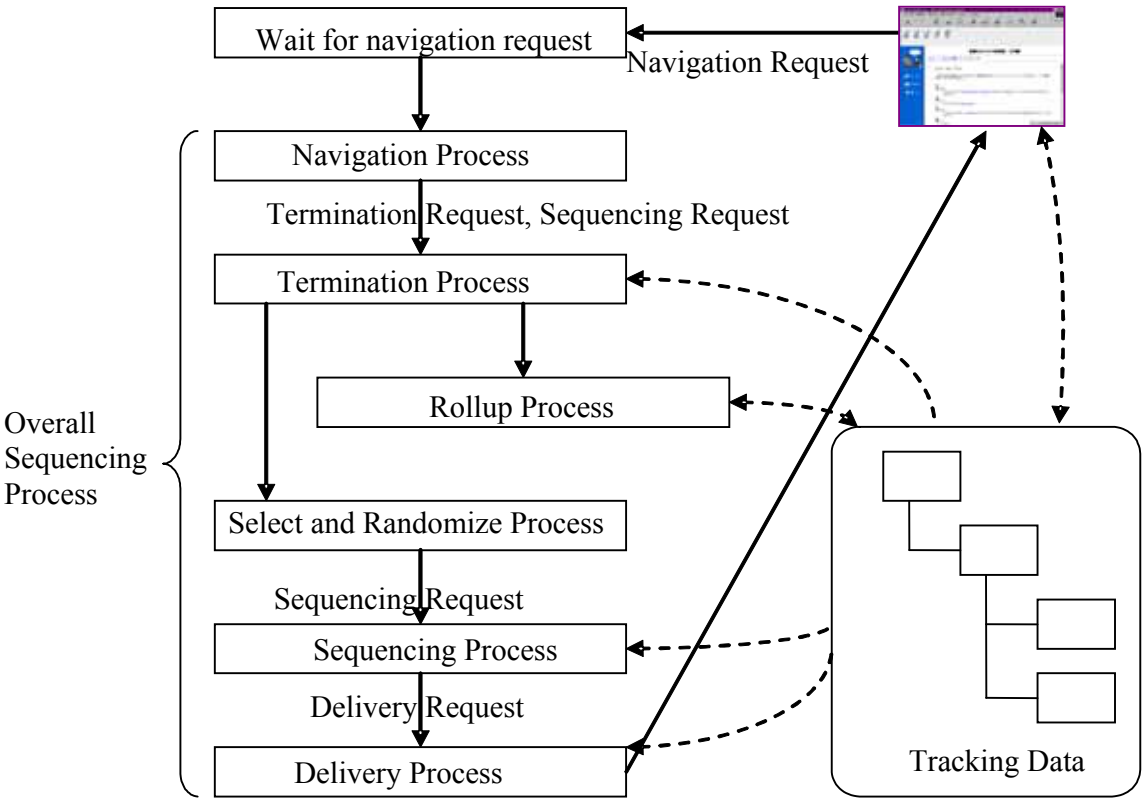


Figure 6.1 Overall Sequencing Process

The sequencing behavior model is implemented through the overall sequencing process. The overall sequencing process starts with a navigation request from the learner's Web browser, determines the next activity to be delivered to the Web browser, and then waits for the next navigation request and repeats this action.

A navigation request is sent from the Web browser. The navigation process judges whether the navigation request is valid, and generates a termination request and a sequencing request as explained in Section 3.3.

The termination process that handles the generated termination request performs such actions as terminating the activity, terminating the whole learning process or suspending the process. After terminating the activity and determining the necessary tracking information, the termination process calls the rollup process that updates the tracking information for the entire activity tree based on the specified rollup rules. After the rollup, defined post-condition rules (if any) are evaluated, and if the conditions are satisfied, the sequencing request from the learner is transformed into another sequencing request.

After this, the selection and randomization process changes the order of activities to be presented if required.

The sequencing request is then passed to the sequencing process which decides the next activity to be delivered. The sequencing process applies different subprocesses, depending on each sequencing request, to determine the next activity to be delivered for the learner by evaluating sequencing control modes, limit conditions and precondition rules, among other things.

Finally, the information concerning the activity that has been identified for delivery is passed to the delivery process as a delivery request. The delivery process reconfirms the limit condition and precondition rules for the activity, initializes the data necessary for tracking the activity, and then delivers the activity.

Each component process of the overall sequencing process may refer to or update the tracking information. The SCO in the Web browser updates the tracking information for the associated leaf activity through the Run-Time Environment. Conversely, the SCO in the Web browser can refer to its own tracking information regarding the associated leaf activity. The rollup process updates the tracking information for the entire activity tree based on the tracking information collected from the related leaf activities. The termination process, the sequencing process and the delivery process evaluate the sequencing rules and limit conditions by referring to the tracking information updated by the rollup process.

6.1.1 Navigation Process

The navigation process waits for a navigation request from the learner's Web browser, and generates a termination request and a sequencing request when it receives a request, as explained in Section 3.3. The navigation process ignores irrelevant navigation requests like a start navigation request when the learning process is already in progress. In the other cases, the navigation process transforms the navigation request into a termination request and a sequencing request as explained in Section 3.3.

6.1.2 Termination Process

The termination process receives a termination request from the navigation process, and terminates the current activity. Depending on the type of termination request, it performs one of the following termination actions:

- *Exit, Exit All*: The values of the run-time data model elements for the SCO are reflected in the tracking information for the associated activity. The current

activity is terminated. In the case of an *Exit All* termination request, the whole learning process is terminated.

- *Abandon, Abandon All*: The values of the run-time data model elements for the SCO are not reflected in the tracking information for the associated activity. The current activity is terminated. In the case of an *Abandon All* termination request, the whole learning process is terminated.
- *Suspend All*: The current activity and all its ancestral activities are suspended. A suspended activity can be resumed with a *Resume All* sequencing request. The attempt in this case is not a new one but a continuation of the attempt interrupted by the suspension.

In the case of an *Exit* termination request, the termination process performs the following actions:

(1) The termination process terminates the current activity, and reflects the values of the Run-Time Environment Data Model elements managed by the SCO during the run-time in the tracking information for the associated activity. If the learning resource associated with the activity is an asset which does not send run time environment information, the LMS sets the default tracking information.
(2) It calls the rollup process, which is explained in the next section.
(3) It evaluates the termination rules and post-condition rules. If the parent activity is to be terminated with an *Exit Parent* or *Exit All* rule, it recursively evaluates the termination rules and post-condition rules for the parent activity. When a sequencing request is generated as a result of a *Continue*, *Previous* or *Retry* rule being applied, the termination process replaces the sequencing request from the navigation process with this sequencing request.

6.1.3 Rollup Process

During the rollup process, the measure rollup, objective rollup and progress status rollup actions are performed. See Section 3.4.5 for the details of these actions.

(1) In the measure rollup, the normalized objective measure for a parent activity is calculated from the weighted values of the measures for the primary objectives of its child activities.
(2) In the objective rollup process, the objective status of a parent activity's primary objective is determined using one of the methods described below. When 1) is applied, the other methods are not applied, and if 2) is applied then 3) is not applied.
    1) Compare the objective measure computed by the measure rollup process against the predefined threshold value.
    2) Apply the defined rollup rules.
    3) Apply the default rollup rules.
(3) In the progress status rollup process, the attempt completion status for a parent activity is determined using one of the following methods: When 1) is applied, 2) is not applied.
    1) Apply the defined rollup rules.
    2) Apply the default rollup rules.

6.1.4 Selection and Randomization Process

The selection and randomization process defines how the child activities of a cluster are selected for presentation to the learner and how the activities are re-ordered for presentation to the learner.

(1) In the selection process, a specified number of candidate activities are selected from among the child activities of a cluster for presentation to the learner. The selection is done when an attempt is made for the first time on the associated cluster.

(2) In the randomization process, the order of presentation for more than one child activity included in a cluster is changed at random. It is possible to specify whether randomization is performed at the first attempt on the cluster or at each new attempt.

6.1.5 Sequencing Process

The sequencing process receives a sequencing request from the navigation process, and determines the next activity to be presented to the learner based on the information in the request. The sequencing process consists of a number of specific processes corresponding to the types of sequencing request. Each process applies sequencing control modes, limit conditions and precondition rules when it determines the next activity to be delivered.

6.1.5.1 Activity Decision Processing Overview

Processes to select the activity to be delivered can be categorized depending on the types of sequencing request.

(1) **Launch**: *Start, Resume All* and *Choice*
These sequencing requests are processed for the start of a learning experience from a state where no current activity has been identified. For a *Start* sequencing request, the *Flow* subprocess (explained below) traverses the target activity tree to determine the next activity to be delivered. For a *Resume All* sequencing request, the learning experience is resumed from the activity that was previously suspended. For a *Choice* sequencing request, a selected activity is delivered for the learner.

(2) **Activity Tree Traversal**: *Continue, Previous* and *Choice*
For these sequencing requests, a target activity is determined by traversing the associated activity tree forward (*Continue*) or backward (*Previous*). The *Flow* subprocess is applied to the traversal of the activity tree. For a *Choice* sequencing request, the selected activity becomes the destination of the movement. If a non-leaf activity is selected, then a target activity is determined by applying the *Flow* subprocess to the cluster consisting of the selected activity and its descendant activities.

(3) *Retry*
For a *Retry* sequencing request, the current activity is again selected for delivery to the learner. If a non-leaf activity is specified in this case, then a target leaf activity is determined by applying the *Flow* subprocess.

(4) *Exit*
For an *Exit* sequencing request, the current attempt on the activity is terminated. If the current activity is the root of an activity tree, the sequencing session ends.

6.1.5.2 Flow Subprocess Overview

The *Flow* subprocess traverses an activity tree from a designated activity in a specified direction to select an activity to be delivered next. A candidate activity to be delivered to the learner is a leaf activity, and if it cannot reach a leaf activity, the *Flow* subprocess returns an error. Below is an outline of the behavior of the *Flow* subprocess:

(1) It moves to the next activity in the activity tree from the current activity in a designated direction, and that activity becomes the candidate activity.
(2) If the *Sequencing Control Flow* of the candidate's parent activity is evaluated as false, then it treats it as an error and terminates the sequencing process.
(3) If the *Skip* precondition rule for the candidate activity is evaluated as true, then it moves from the candidate to another activity in the designated direction, selects that activity as a candidate, and returns to (2) above.
(4) If the *Disable* precondition rule or the limit condition for the candidate activity is evaluated as true, then it raises an error state and terminates the sequencing process.
(5) If the candidate is a leaf activity, then it selects this as the next activity to be delivered, and terminates the sequencing process.
(6) If the candidate is not a leaf activity, it enters the cluster and selects a candidate.
In the case of forward traversal, the first child activity of the cluster becomes a candidate.
In the case of backward traversal where the *Sequencing Control Forward Only* is false, the last child activity of the cluster becomes a candidate.
In the case of backward traversal where the *Sequencing Control Forward Only* is true, the first child activity of the cluster becomes a candidate.

6.1.6 Delivery Process

The delivery process receives a delivery request for the candidate activity that has been selected through the sequencing process, and delivers the activity to the client side. During the delivery process, whether the limit conditions are violated is checked for that candidate activity and all its ancestral activities. An attempt on the activity is then started and the logging of the elapsed time for the activity and other data is started as well.

The actual learning resource, an SCO or asset, is delivered using the launch process of the Run-Time Environment.

6.2 Pseudo-Code

SCORM 2004 provides a precise specification for the behaviors of the component processes that constitute the overall sequencing process. This section describes the relationships between these process calls and provides an overview of their actions.

Tables 6.1 and 6.2 show the relationships between process calls. Table 6.1 shows the relationships between process calls used in the overall sequencing process. The processes that are called from more than one place are highlighted in bold letters. These processes are explained in detail in Table 6.2. The numbers in a pair of square

brackets are the process identification codes used in the pseudo-codes of the standard specifications.

A summary of the major processes is given below.

Table 6.1 Pseudo-Code Process Call Relationships

Overall Sequencing Process [OP.1]
Navigation Request Process [NB.2.1]
Termination Request Process [TB.2.3]
End Attempt Process [UP.4]
Sequencing Exit Action Rules Subprocess [TB.2.1]
Sequencing Rules Check Process [UP.2]
Terminate Descendent Attempts Process [UP.3]
End Attempt Process [UP.4]
Sequencing Post Condition Rules Subprocess [TB.2.2]
Sequencing Rules Check Process [UP.2]
Terminate Descendent Attempts Process [UP.3]
Select Children Process [SR. 1]
Randomize Children Process [SR.2]
Sequencing Request Process [SB.2.12]
Start Sequencing Request Process [SB.2.5]
Flow Subprocess [SB.2.3]
Resume All Sequencing Request Process [SB.2.6]
Exit Sequencing Request Process [SB.2.11]
Retry Sequencing Request Process [SB.2.10]
Flow Subprocess [SB.2.3]
Continue Sequencing Request Process [SB.2.7]
Flow Subprocess [SB.2.3]
Previous Sequencing Request Process [SB.2.8]
Flow Subprocess [SB.2.3]
Choice Sequencing Request Process [SB.2.9]
Sequencing Rules Check Process [UP.2]
Choice Activity Traversal Subprocess [SB.2.4]
Sequencing Rules Check Process [UP.2]
Choice Flow Subprocess [SB.2.9.1]
Choice Flow Tree Traversal Subprocess [SB.2.9.2]
Choice Flow Tree Traversal Subprocess [SB.2.9.2] REC
Flow Subprocess [SB.2.3]
Terminate Descendent Attempts Process [UP.3]
Delivery Request Process [DB.1. 1]
Check Activity Process [UP.5]
Content Delivery Environment Process [DB.2]
Clear Suspended Activity Subprocess [DB.2.1]
Terminate Descendent Attempts Process [UP.3]

6.2.1 Overall Sequencing Process

This is the process that covers the entire set of sequencing behaviors. The processes making up the overall process are called in sequence to perform the actions discussed in Section 6.1. That is, the sequencing actions are performed by calling, one after another, the Navigation Request Process, the Termination Request Process, the Select Children Process, the Randomize Children Process, the Sequencing Request Process, the Delivery Request Process and the Content Delivery Environment Process.

Table 6.2 Pseudo-Code Process Call relationships (Subprocesses)

Flow Subprocess [SB.2.3]
Flow Tree Traversal Subprocess [SB.2.1]
Flow Tree Traversal Subprocess [SB.2.1] REC
Flow Activity Traversal Subprocess [SB.2.2]
Sequencing Rules Check Process [UP.2]
Flow Tree Traversal Subprocess [SB.2.1]
Flow Activity Traversal Subprocess [SB.2.2] REC
Check Activity Process [UP.5]

Terminate Descendent Attempts Process [UP.3]
End Attempt Process [UP.4]

End Attempt Process [UP.4]
Overall Rollup Process [RB.1. 5]
Measure Rollup Process [RB.1.1]
Objective Rollup Process [RB.1.2]
Objective Rollup Using Measure Process [RB.1.2 a]
Objective Rollup Using Rules Process [RB.1.2 b]
Rollup Rule Check Subprocess [RB.1.4]
Activity Progress Rollup Process [RB.1.3]
Rollup Rule Check Subprocess [RB.1.4]

Rollup Rule Check Subprocess [RB.1.4]
Check Child for Rollup Subprocess [RB.1.4.2]
Sequencing Rules Check Process [UP.2]
Evaluate Rollup Conditions Subprocess [RB.1.4.1]

Check Activity Process [UP.5]
Sequencing Rules Check Process [UP.2]
Limit Conditions Check Process [UP.1]

Sequencing Rules Check Process [UP.2]
Sequencing Rule Check Subprocess [UP.2.1]

6.2.2 Termination Request Process

The *Termination Request Process* processes termination requests. It consists of the *End Attempt Process*, which ends the attempt on the current activity and performs rollup, the *Sequencing Exit Action Rules Subprocess*, which processes the exit rules, the *Sequencing Post Condition Rules Subprocess*, which processes post-condition rules, and the *Terminate Descendent Attempts Process*, which performs the termination of the ancestral activities of the current activity.

6.2.3 Sequencing Request Process

Depending on the received sequencing request, a specific process is called from the following sequencing request subprocesses: *Resume All*, *Exit*, *Retry*, *Continue*,

Previous and *Choice*. Among these processes, the *Start*, *Retry*, *Continue*, and *Previous* sequencing request subprocesses apply the *Flow Subprocess* for an activity tree traversal to select the activity to be delivered.

In the case of the *Choice* sequencing request subprocess, it is possible to determine whether the candidate is valid for delivery by applying a specific traversal, depending on the relationship between the current activity and the activity tree containing the candidate activity.

6.2.4 Flow Subprocess

The *Flow* subprocess determines a candidate activity for delivery by traversing the activity tree forward or backward. Its internal structure is designed in such a way that the *Flow Tree Traversal Subprocess* and the *Flow Activity Traversal Subprocess* are recursively called. In the *Flow Tree Traversal Subprocess*, the activity next to the current activity in the designated direction becomes a candidate without the precondition rules and others being checked. In the *Flow Activity Traversal Subprocess*, the precondition rules and other conditions are evaluated for the candidate activity determined by the *Flow Tree Traversal Subprocess*, and whether the candidate is valid for delivery is checked. If the *Skip* precondition rules are evaluated as true, the *Flow Tree Traversal Subprocess* and the *Flow Activity Traversal Subprocess* are recursively called in the traversal of the activity tree.

6.2.5 End Attempt Process

The *End Attempt Process* performs the end processing of the activity. Specifically, when the activity is a leaf activity and there is no SCO associated with the activity, the tracking information for the activity is reset with default values. The overall sequencing process is called and rollup is performed. In the *Overall Rollup Process*, the *Measure Rollup Process*, the *Objective Rollup Process* and the *Activity Progress Rollup Process* are called to perform, respectively, measure rollup, objective rollup and progress rollup actions.

6.2.6 Check Activity Process

The *Check Activity Process* checks the limit conditions, the *Disable* precondition rules and other conditions for the activity, and confirms whether the activity is valid for delivery.

6.2.7 Sequencing Rule Check Process

This process checks whether the condition clause of a sequencing rule is evaluated as true.

7 Implementing the Run-Time Environment

As discussed in Section 5, SCORM specifies the Run-Time Environment for communication between the LMS and SCOs. This section describes how to implement the SCORM Run-Time Environment, and the points to be noted during the implementation, from the LMS implementer's standpoint. This section explains how to implement the launching and ending process of an SCO, how to implement an API Instance, and then how to deal with both SCORM 1.2 compliant SCOs and SCORM 2004 compliant SCOs. In addition, it discusses the points to be noted in implementing the RTE Data Model elements, the relationships between the data model elements and attempts, and data retention issues for handling attempts that are suspended and then resumed, and so on. This section also covers the relationship of sequencing with tracking information, and then discusses the points to be noted in implementing the navigation function.

7.1 Launch

In the launch process, the LMS selects a learning resource to be delivered to the learner and launches it. If the delivered resource is an SCO, the SCO searches for the API Instance that is provided by the LMS, and maintains a communication session with the LMS using the API Instance. This subsection explains a sample implementation of an LMS to deal with this process.

Figure 7.1 shows a screenshot of a Web browser through which a learner is studying under a SCORM compliant LMS. In this example, the browser screen is divided into three frames. A set of command buttons for issuing navigation requests is located in the Menu frame (top). What is displayed in this frame varies depending on the LMS. The SCO frame (middle) displays the learning resource. What is displayed in this frame is the learning resource itself. The bottom frame is where the API Instance is located, and the size of this frame is set to 0 so that learners cannot see it.

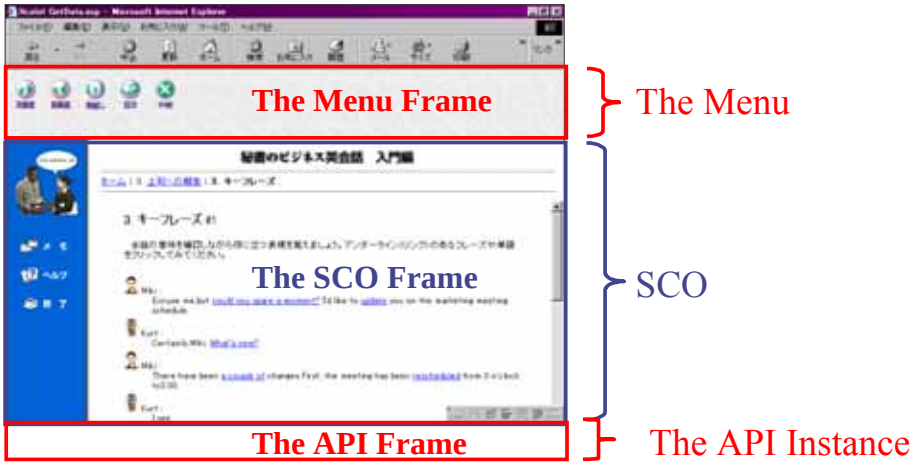


Figure 7.1 Frame Composition of a Client Window

The LMS implements the launching process by assigning an API Instance on the Web browser and sending the designated resource to the Web browser. This is done by

sending an HTML file in which the three frames are defined. An example of such an HTML file is shown below:

```
<HTML>
<HEAD>
<TITLE>LMS system</TITLE>

<SCRIPT LANGUAGE ="JavaScript">
   function myOnLoad( url )                                    (1)
   { self.Main.location = url; }
</SCRIPT></HEAD>

<FRAMESET ROWS="100%,0%" onLoad = myOnLoad("sco_url")>         (2)
   <FRAMESET ROWS="43,*">                                      (3)
          <FRAME NAME="Menu" SRC="menu_generation_url">
          <FRAME NAME="Main" SRC="about:blank">                (4)
   </FRAMESET>
   <FRAME NAME="API_1484_11" SRC="api_instance_url">            (5)
</FRAMESET>

</HTML>
```

As shown in (2) and (3) above, the menu frame (name: Menu), the SCO frame (name: SCO) and the API Instance frame (name: API_1484_11) are defined. Note that the menu frame and the SCO frame are defined as one step deeper than the API Instance frame. This is because the SCO has to search for the API Instance starting from its parent frame as explained in Section 5.3.3.

There is an important point to be noted in this HTML definition. The SCO frame is initially left blank in (4), and when the frame is loaded, the *onLoad* event is triggered and thus *myOnLoad* function (1) is invoked to actually download the SCO. If the URL of the SCO is directly specified in (4) without using this mechanism, whether the SCO or the API Instance will be downloaded to the Web browser first is uncertain. If the SCO is downloaded first and starts searching for the API Instance, the SCO may fail to find the API Instance. As this is a timing-related problem, it will be very difficult to detect because such a problem may occur on some Web browsers but not on others. To prevent such a problem, the *onLoad* event (2) is used in the above HTML definition to invoke the *myOnLoad* function (1), which will actually download the SCO. As the *onLoad* event is triggered after the frameset definition has been completely interpreted, the SCO is always downloaded (1) only after the API Instance (5) has been downloaded. Figure 7.2 shows this temporal relationship.

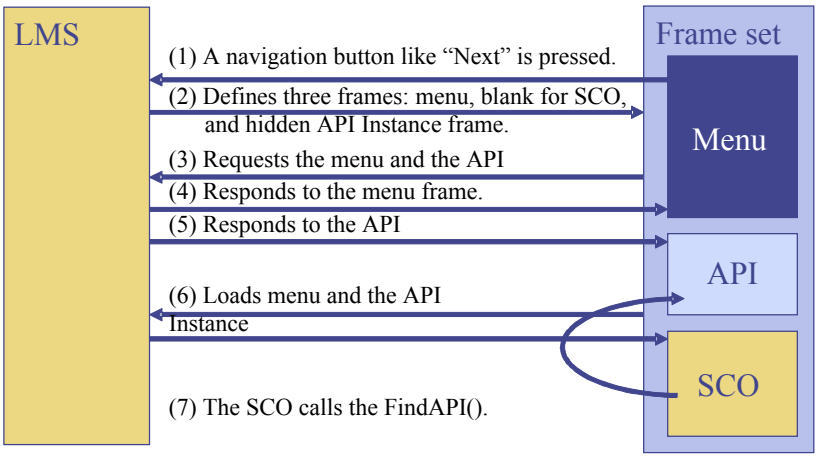


Figure 7.2 Temporal Relationships at SCO Launching Time

7.2 Implementing the API Instance

An API Instance is implemented in the form of a DOM object. In this DOM object, the API methods are defined in the ECMAScript language, which is generally known as JavaScript. The only requirement specified in the standard for an API implementation is that it enable communication of the run-time data model elements to the LMS using the API methods specified in the standard. The other details are left to the implementers of specific LMS products. For example, it is not specified in the standard whether the values of the data model elements should be kept on the client or the server. In general, the simplest solution may be to keep the business logic for handling data elements only on the server rather than on the client, but when the system performance is considered important, it may be better to allow the client to manage data model elements as much as possible from the viewpoint of communication volume and the server processing load.

The standard specifies that the API Instance be named "API_1484_11". However, the name was "API" in SCORM 1.2. By taking advantage of this difference, it is possible to make both SCORM 1.2 compliant and SCORM 2004 compliant SCOs run under a SCORM 2004 compliant LMS. Figure 7.3 shows how this can be done. In this example, the LMS provides two separate API Instances under the names of "API_1484_11" and "API". As a SCORM 2004 compliant SCO will use the former API Instance while a SCORM 1.2 compliant SCO will use the latter API Instance, there is no need for the SCOs to be conscious of the difference in API method names and data model element names between SCORM 2004 and SCORM 1.2. The SCORM 1.2 API Instance is implemented in such a way that the run time data model elements and API methods of SCORM 1.2 are converted to those of SCORM 2004. Refer to Section 8 for details.

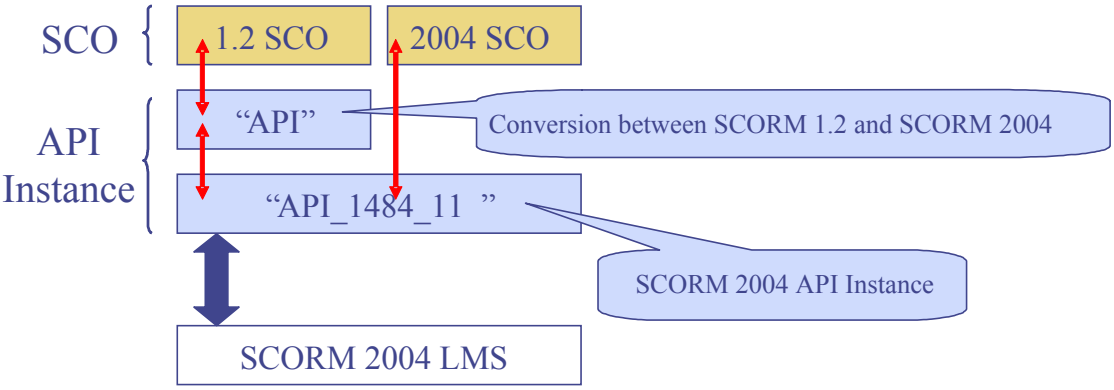


Figure 7.3 Running both SCORM 1.2 and SCORM 2004 SCOs under a SCORM 2004 LMS

7.3 Implementing Data Model Elements

From the viewpoint of an SCO, the data model elements provided in the Run-Time Environment have three modes of communication direction: *read-only*, *write-only* and *read / write*. Among these data model elements, most of the read-only data model elements are initialized by the LMS from the management data and the information provided in the manifest file so that SCOs can use them. Therefore, the LMS must initialize these. There are data model elements that are not currently used but contain fixed values. In the case of read / write data model elements, the LMS should preserve the values set by the SCOs, and return the values when the SCOs request them. However, caution is required as some data model elements are related to the tracking model in sequencing. This section explains the data model elements that require caution when they are implemented.

7.3.1 Data Model Elements Initialized by the LMS from Management Data

The *cmi.learner_id* and *cmi.learner_name* data model elements are read-only elements, and they are initialized from management data. In general, as it manages the IDs and names of learners, the LMS makes this information available for SCOs to retrieve through the initialization of the above data model elements.

7.3.2 Data Model Elements Initialized from the Manifest File

The following data model elements are read-only elements and are initialized from the values specified in the manifest file: *cmi.completion_threshold*, *cmi.launch_data*, *cmi.max_time_allowed*, *cmi_scaled_passing_score* and *cmi.time_limit_action*. Table 7.1 shows this correspondence. If an SCO attempts to retrieve the value of a data model element whose value is not specified in the manifest file, an error is raised with "*403 – Data Model Element Value Not Initialized*."

Table 7.1 Data Model Elements Initialized from the Manifest File

Data Model Element Name Manifest Element Name	Description
cmi.completion_threshold *<adlcp:completionThreshold>*	Threshold performance measure for determining the completion of the SCO
cmi.launch_data *<adlcp:dataFromLMS>*	Initialization parameters for launching the SCO
cmi.max_time_allowed *<imsss:attemptAbsoluteDurationLimit>*	The maximum amount of accumulated time allowed for one attempt on the SCO
cmi.scaled_passing_score *<imsss:minNormalizedMeasure>*	The scaled passing score required to be considered to have mastered the SCO
cmi.time_limit.action *<adlcp:timeLimitAction>*	The action to be taken by the SCO when the *max_time_allowed* value is reached

7.3.3 Data Model Elements with Fixed Values

An LMS is required to implement *cmi.credit* and *cmi.mode* as read-only data model elements. However, the uses of these elements are not specified. It will be sufficient to set "*credit*" and "*normal*" to these data model elements, respectively.

7.3.4 Read-only and Write-Only Data Model Elements Related to Each Other

The *cmi.session_time* and *cmi.total_time* pair and the *cmi.exit* and *cmi.entry* pair of data model elements are each write-only and read-only element pairs, but the values of the read-only data model elements are affected by the values of the write-only data model elements.

The *cmi.session_time* data model element represents the elapsed session time from the launching of the SCO, and the *cmi.total_time* data model element represents the total sum of a learner's session times on the SCO. The LMS sums the values of the last *cmi.session_time* data model elements reported from the launching of an SCO until its termination, and sets the value to the *cmi.total_time* data model element. If a *cmi.session_time* value is reported during a session, it will be ignored by the LMS. The initial value of the *cmi.total_time* is 0.

The *cmi.exit* data model element represents information regarding how and why the learner left the SCO, and the *cmi.entry* data model element indicates the exit status of the SCO in the previous attempt. The initial value of *cmi.entry* is "*ab_initio*"; the LMS sets the value of *cmi.entry* to "*resume*" if the *cmi.exit* value is "*suspend*" or "*logout*", and sets it to "" if the value of *cmi.exit* is "*normal*" or "*time-out*".

When an attempt is resumed (*Resume All*) after it was suspended (*Suspend All*), the *cmi.total_time* and *cmi.entry* data model elements should reflect the values prior to the interruption. Therefore, the LMS should store the values of *cmi.total_time* and *cmi.entry* using a persistent file or other means for each SCO, and restore them at the time when the attempt is resumed.

7.3.5 Data Model Elements that Must be Preserved and Restored

The following data model elements are read / write type elements: *cmi.location, cmi.suspend_data, cmi.interactions, cmi.learner_preference, cmi.score, cmi.progress_measure, cmi.objectives, cmi.completion_status,* and *cmi.success_status*. The LMS must ensure that the values of these data model elements set by an SCO are preserved so that they can be retrieved at a later stage. If a learner has suspended an attempt on an SCO, the LMS must store the values prior to the interruption using a persistent file or other means so that those values can be restored when the attempt is resumed.

For *cmi.location, cmi.suspend_data* and *cmi.interactions*, it is necessary to ensure that the values set by the SCO are allowed to be retrieved as they are. The LMS should not process these data elements. If the value of one of these data elements is queried before the SCO writes a value to it, an error is raised with "*403 – Data Model Element Value Not Initialized.*"

The *cmi.learner_preference* data model element is a read / write element for keeping the learner's specific preferences, such as the language that the learner will use, the volume of the speakers and so on, with the SCO. This element may be initialized by the LMS using the management data, and then may be updated by the learner during a learning session. If the LMS does not maintain relevant management data, a specific default value to each data model element is set.

In addition, for *cmi.score*, *cmi.progress_measure*, and *cmi.objectives,* it is necessary to ensure that the values set by the SCO are allowed to be retrieved as they are. The LMS should not process these data elements. If the value of one of these data elements is queried before the SCO writes a value to it, an error is raised with "*403 – Data Model Element Value Not Initialized.*"

The values of the *cmi.completion_status* and *cmi.success_status* data model elements are used as reported by the SCO. The initial values of the elements are "*Unknown*", but the LMS may determine their values by comparing the values of *cmi.progress_measure* and *cmi.score_scaled* against the values defined in the manifest file (*adlcp:completionThreshold* and *imsss:minNormalizedMeasure*).

Tables 7.2 and 7.3 respectively show the rules for determining the values of *cmi.completion_status* and *cmi.success_status*.

Table 7.2 Determining the Completion Status

Manifest's *adlcp:completion Threshold* = *cmi.completion_ threshold*	*cmi.progress_ measure* set by SCO	*cmi.completion_ status* set by SCO	*cmi.completion_ status* set by LMS
unknown	*unknown*	*unknown*	*unknown*
unknown	*unknown*	Token（Note）	Value by SCO
unknown	*0.5*	*unknown*	*unknown*
unknown	*0.5*	Token	Value by SCO
0.8	*unknown*	*unknown*	*unknown*
0.8	*unknown*	Token	Value by SCO
0.8	*0.5*	*unknown*	*incomplete*
0.8	*0.5*	Token	incomplete
0.8	*0.9*	*unknown*	*completed*
0.8	*0.9*	Token	*completed*

Note: One of the three vocabulary tokens: *completed*, *incomplete,* or *not attempted*

Table 7.3 Determining the Success Status

Manifest's *imsss:min Normalized Measure* = *cmi.scaled_ passing_score*	*cmi.score. scaled* set by SCO	*cmi.success_stat us* set by SCO	*cmi.success_stat us* set by LMS
unknown	*unknown*	*unknown*	*unknown*
unknown	*unknown*	Token（Note）	Value by SCO
unknown	*0.5*	*unknown*	*unknown*
unknown	*0.5*	Token	Value by SCO
0.8	*unknown*	*unknown*	*unknown*
0.8	*unknown*	Token	Value by SCO
0.8	*0.5*	*unknown*	*failed*
0.8	*0.5*	Token	*failed*
0.8	*0.9*	*unknown*	*passed*
0.8	*0.9*	Token	*passed*

Note: One of the two vocabulary tokens: *passed* or *failed*

The basic way to determine the completion status is as follows:

- If the *cmi.progress_measure* value is greater than or equal to the *cmi_completion_threshold* value, the status is *completed*. The value reported by the SCO for *cmi.completion_status* is ignored.
- If the *cmi.progress_measure* value is smaller than the *cmi_completion_threshold* value, the status is *incomplete*. The value reported by the SCO for *cmi.completion_status* is ignored.
- If either the *cmi.progress_measure* value or the *cmi_completion_threshold* value is *unknown*, the completion status is determined by the value reported by the SCO for *cmi.completion_status*.

The value of *cmi.success_status* is determined in a similar way by giving a higher priority to the comparison of the *cmi.score_scaled* value against the *cmi.scaled_passing_score* value. If one of the data model element values is *unknown*, the value reported by the SCO is applied.

By taking advantage of this behavior to determine the success status, it is possible to change the conditions for passing by associating one SCO with more than one activity and then specifying different passing scores for these activities within the manifest file.

Note that the *cmi.completion_status*, *cmi.success_status*, *cmi.score.scaled*, *cmi.progress_measure*, *cmi.objectives.n.success_status*, and *cmi.objectives.n.score.scaled* data model elements are related to the tracking information for the sequencing function. These are explained in the next section.

7.4 Tracking Model and RTE Data Model

For the Tracking Model elements described in Section 3.2, the information about the completion of a leaf activity and the objective progress information is set by using RTE Data Model elements. If the learning resource associated with an activity is an asset rather than an SCO, the LMS sets pre-determined values to these elements.

7.4.1 Setting Information Regarding Activity Completion

Table 7.4 shows the correspondence between the Tracking Model elements and the RTE Data Model elements related to the completion of an activity.

The value of the *Attempt Completion Status* element is determined by the value of *cmi.completion_status*. The value of *cmi.completion_status* is generally set with a value reported by the SCO, but as was discussed in a previous section, caution is required because the *cmi.completion_status* value is sometimes determined by the LMS by comparing the value of *cmi.progress_measure* against that of *cmi.completion_threshold*.

As the value of *cmi.completion_status* will not be set when the learning resource associated with the current activity is an asset rather than an SCO, the LMS sometimes automatically sets the value of *cmi.completion_status* to "*completed*". The LMS displays this behavior under the following conditions:

- If the *Tracked* element of the activity is set to *True*, and
- If the *Completion Set by Content* element is set to *False*, and
- If the value of *cmi.completion_status* is *Unknown*.

Although *Attempt Completion Amount* and *cmi.progress_measure* are defined as data model elements, these are not currently used in sequencing and their relationship is not specified.

Table 7.4 Tracking Model and RTE Data Model Elements for Attempt Completion

RTE Data Model Elements		Tracking Model Elements	
cmi.completion_status		*Attempt Completion Status*	
	completed		*completed* (Initial value for assets)
	incomplete		*incomplete*
	not attempted		*incomplete*
	unknown		*unknown* (Initial value)
Note: See Table 7.2 for how to determine the value of *cmi.completion_status*.			
cmi.progress_measure		*Attempt Completion Amount*	
	[0..1]		*[0..1]*
	unknown		*unknown* (Initial value)

Note that SCORM 2004 does not use *Attempt Completion Amount* for sequencing, and it does not specify the relationship of this Tracking Model element with *Progress Measure*.

7.4.2 Setting Objective Progress Information for the Primary Objective

Table 7.5 shows the correspondence between the Tracking Model and the RTE Data Model elements.

The progress information about an objective is managed with the *Objective Progress Status* and *Objective Normalized Measure* elements. In the case of the primary objective, these values are determined by the values of *cmi.success_status* and *cmi.score.scaled*. In principle, the values of these RTE Data Model elements are honored as they are reported by each SCO; however, as shown in Table 7.3, the value of *cmi.success_status* is sometimes determined by the LMS by comparing the *cmi.score_scaled* value against the *cmi.scaled_passing_score* value.

As the value of *cmi.success_status* will not be set when the learning resource associated with the current activity is an asset rather than an SCO, the LMS may sometimes automatically set the value to "*completed*". The LMS displays this behavior under the following conditions:

- If the *Tracked* element of the activity is set to *True*, and
- If the *Objective Set by Content* element is set to *False*, and
- If the value of cmi.success_status is *Unknown*.

Table 7.5 Tracking Model and RTE Data Model Elements for Progress Information about the Primary Objective

RTE Data Model Elements		Tracking Model Elements	
cmi.success_status		*Objective Satisfied Status*	
	passed		*satisfied*
	failed		*not satisfied*
	unknown		*unknown* (Initial value)
`cmi.score_scaled`		`Objective Normalized Measure`	
	[-1..1]		*[-1..1]*
	unknown		*unknown* (Initial value)

Note: Refer to Table 7.3 for how to determine the value of *cmi.success_status*.

7.4.3 Objective Progress Information for Objectives other than the Primary Objective

The *Objective Satisfied Status* element values for other objectives are determined by *cmi.objectives.n.success_status* and the *Objective Normalized Measure* values are determined by *cmi.objectives.n.score_scaled*. Between *cmi.objectives.n.success_status* and *cmi.objectives.n.score_scaled* in this case, however, there is no dependency relationship as there is with *cmi.success_status* and *cmi.score.scaled* for the primary objective. For these objectives, furthermore, the LMS does not perform an automatic initialization even if the resource associated with the activity is an asset. The value of *cmi.objectives.n.id* that can be retrieved by an SCO is an ID (Activity Objective ID) assigned to a local objective for the activity.

When a local objective is connected to a shared global objective in such a way that the local objective reads the value of the shared global objective, the values of *cmi.objectives.n.success_status* and *cmi.objectives.n.score.scaled* that the SCO can retrieve are the same as those of the *Objective Satisfied Status* and *Objective Normalized Measure* for the shared global objective. On the other hand, when a local objective is connected to a shared global objective in such a way that the local objective writes the value of the shared global objective, the values of *cmi.objectives.n.success_status* and *cmi.objectives.n.score.scaled* that the SCO will send are reflected in the values of the *Objective Satisfied Status* and *Objective Normalized Measure* for the shared global objective.

Table 7.6 Tracking Model and RTE Data Model Elements for Objective Progress Information for Objectives other than the Primary Objective

RTE Data Model Elements		Tracking Model Elements	
cmi.objectives.n.success_status		*Objective Satisfied Status*	
	passed		*satisfied*
	failed		*not satisfied*
	unknown		*unknown* (Initial value)
cmi.objectives.n.score_scaled		*Objective Normalized Measure*	
	[-1..1]		*[-1..1]*
	unknown		*unknown* (Initial value)
cmi.objectives.n.id		*Activity Objective ID*	
	Universal Resource Identifier		Unique identifier

7.5 Implementing the Navigation Function

As explained in Section 4, SCORM 2004 allows an SCO to trigger navigation events. The SCO can trigger a navigation event by calling the RTE API method *SetValue("adl.nav.request",* <REQUEST>*).* If the SCO invokes the *Terminate("")* method after that, the LMS processes the navigation request indicated by <REQUEST>. Although the SCO can invoke *SetValue("adl.nav.request",* <REQUEST>*)* any number of times, the LMS ignores all the calls except the one immediately before the *Terminate("")* API method call. The <REQUEST> in the above SetValue function call must be one of the following: *"continue"*, *"previous"*, *"choice"*, *"exit"*, *"exitAll"*, *"abandon"*, *"abandonAll"* or *"_none*". If the last

navigation request before the *Terminate("")* method is the one with *"_none"*, the LMS does not do anything.

Navigation events are triggered not only by SCOs but also by the navigation user interface devices provided by the LMS. For this reason, there can be a situation where a navigation event triggered by the SCO is in competition with one triggered by a user interface device provided by the LMS. In this case, the navigation event triggered by the LMS-provided user interface device is always selected for processing with a higher priority.

Suppose an SCO has invoked the *SetValue("adl.nav.request", "continue")* method call and then the learner has chosen another SCO from the menu displayed by the LMS with a *"choice"* navigation request. In this case, the new SCO is loaded onto the Web browser and the old SCO is unloaded. As an SCO generally calls the *Terminate("")* API method at the unload event, the *"continue"* navigation event that was triggered earlier competes with the *"choice"* navigation request triggered by the learner. In this case, the LMS gives priority to the navigation event issued by the learner and ignores the *"continue"* navigation event.

8 Migration from SCORM 1.2 to SCORM 2004

Compared with SCORM 1.2, SCORM 2004 has added sequencing and navigation features and introduces a more detailed specification. This section describes the differences between the two and how to convert contents to the new standard.

8.1 Differences in the Manifest File and Migration Methods

In principle, SCORM 2004 compliant Manifest files are upwardly compatible with SCORM 1.2 Manifest files. In a SCORM 2004 environment, a SCORM 1.2 manifest file is regarded as a SCORM 2004 Manifest file which does not contain any sequencing or navigation rule definitions.

Note, however, that a special caution is required as the following SCORM 1.2 elements have been deprecated in SCORM 2004:

- *adlcp:prerequisites*
- *adlcp:masteryscore*
- *adlcp:maxtimeallowed*

The *adlcp:prerequisites* data model element for an activity was previously defined as a logical expression with the completion status and progress status information of another activity to determine whether a candidate activity was valid for the learner. Such a condition could be defined in SCORM 1.2, but there was no specification as to how the LMS should behave for the possible results of the condition, and this caused interoperability problems. As the behavior of each LMS implementation was not predictable under the previous version, it is necessary to convert the value of this data model element into a set of sequencing rules as designed by the content developer when converting content to SCORM 2004.

The *adlcp:masteryscore* and *adlcp:maxtimeallowed* data model elements are replaced in SCORM 2004 with *imsss:minNormalizedMeasure* and *imsss:attemptAbsoluteDurationLimit*, respectively. The values of these elements can be referred to by an SCO during the run-time via the Run-Time Environment. The corresponding RTE data model elements are *cmi.scaled_passing_score* and *cmi.max_time_allowed*.

SCORM 1.2 compliant Manifest files that do not contain *adlcp:prerequisites*, *adlcp:masteryscore* or *adlcp:maxtimeallowed* definitions can be used under a SCORM 2004 compliant LMS without change. It is also possible to take advantage of the new features of SCORM 2004 by adding sequencing rules into such manifest files as required. The cost of converting manifest files to SCORM 2004 is not expected to be very high.

8.2 Differences in the RTE and Migration

As explained in Section 5, the new Run-Time Environment (RTE) specifications are not compatible with their predecessor as there have been considerable changes made to the name of the API Instance, the API method names, data model elements and so

on. To run SCORM 1.2 compliant SCOs in a SCORM 2004 compliant environment, the SCOs must be modified as required by the changes. The amount of necessary modification is expected to be so large that this may create a bottleneck in the whole migration process.

The following subsections describe the measures to be taken on the LMS side to avoid such a bottleneck. At the same time, SCORM 1.2 compliant LMS products and SCORM 2004 compliant products are expected to coexist for some time in the future. Measures that can be taken on the SCO side to cope with such a situation are also discussed below.

8.2.1 What Should Be Done on the LMS for RTE Migration

To enable SCORM 1.2 compliant SCOs to run in a SCORM 2004 environment, the LMS can take advantage of the difference in the names of the API Instance as explained in Section 7.2. The DOM object name for the SCORM 1.2 API Instance is "*API*" and that for the SCORM 2004 API Instance is "*API_1484_11*". Thus, a SCORM 1.2 compliant SCO searches for an API Instance called "*API*", while a SCORM 2004 compliant SCO searches for an API Instance called "*API_1484_11*". SCORM 1.2 content can be enabled to run in a SCORM 2004 environment by taking advantage of this difference in names.

To deal with this difference, the LMS can be implemented in such a way that it provides two API Instances under the names of "*API*" and "*API_1484_11*" as shown in Figure 7.3. The API Instance named "*API*" exposes the RTE API methods of SCORM 1.2 while the API Instance named "*API_1484_11*" exposes the RTE API methods of SCORM 2004. As the SCOs of the two different standards will then search for an appropriate API Instance and make correct API method calls, there will be no need to change the SCOs.

The SCORM 1.2 API Instance mentioned above should be implemented in such a way that it internally translates the data model element names and error codes between the two SCORM versions. As most of the data model elements are compatible between the two versions, except for a few name changes, it is possible to map one set to another without change. As the *cmi.score.lesson_status* SCORM 1.2 data model element is expanded to *cmi.completion.status* and *cmi.success_status* in SCORM 2004, however, it is necessary to add a conversion rule for handling this difference. Tables 8.1 and 8.2 show examples of the rules for handling such conversions. The same conversion method can be applied for the *cmi.objectives.n.status* SCORM 1.2 data model element and the *cmi.objectives.n.completion.status* and *cmi.objectives.n.success_status* SCORM 2004 data model elements. Note also that *cmi.score.scaled* and *cmi.objectives.n.score.scaled* are newly added in SCORM 2004. Such a measure is needed to ensure that appropriate values are set to these elements from an SCO that affect sequencing. SCORM 1.2 did not have data model elements corresponding to these new elements, but assigning the respective normalized values of *cmi.core.score.raw* and *cmi.objective.n.score.raw* to the new data model elements is considered appropriate. Note that the conversion rules mentioned above are not defined in the standard, and LMS implementers will have to incorporate them as required.

8.2.2 What Should Be Done on SCOs for RTE Migration

The fact that SCORM 2004 has been released as a new standard does not mean that all LMS products will automatically be converted to the standard, and SCORM 1.2 compliant and SCORM 2004 compliant LMS products are expected to coexist for a while. For this reason, it is desirable to develop content in a way that allows it to run under both LMS environments.

Table 8.1 Converting Progress Status Data from SCORM 1.2 to SCORM 2004

SCORM 1.2		SCORM 2004	
cmi.core.lesson_status		*cmi.completion_status*	*cmi.success_status*
unknown	→	*unknown*	*unknown*
passed	→	*completed*	*passed*
failed	→	*incomplete*	*failed*
completed	→	*completed*	*unknown*
incomplete	→	*incomplete*	*unknown*
browsed	→	*incomplete*	*unknown*
not attempted	→	*unknown*	*unknown*

Table 8.2 Converting Progress Status Data from SCORM 2004 to SCORM 1.2

SCORM 1.2		SCORM 2004	
cmi.core.lesson_status		*cmi.completion_status*	*cmi.success_status*
unknown	←	*unknown*	*unknown*
passed	←	*unknown*	*passed*
failed	←	*unknown*	*failed*
completed	←	*completed*	*unknown*
passed	←	*completed*	*passed*
failed	←	*completed*	*failed*
incomplete	←	*incomplete*	*unknown*
passed	←	*incomplete*	*passed*
failed	←	*incomplete*	*failed*

Figure 8.1 shows how to implement such an SCO. The SCO is equipped with a virtual API that deals with the two API Instances. The virtual API determines whether the current environment conforms to SCORM 1.2 or SCORM 2004, and then invokes appropriate API methods for the SCO and performs data model element conversion.

To determine the version of the current Run-Time Environment, for example, the SCO searches for its API Instance, first using the name "*API_1484_11*" and then the name "*API*", and then determines from the name of the found API Instance whether it runs in a SCORM 1.2 environment or a SCORM 2004 environment.

For the translation of API method calls and data model elements between the two standards, the virtual API provides the SCO with functions corresponding to the RTE API methods so that when these functions are called, they are translated into their corresponding API methods of the current Run-Time Environment.

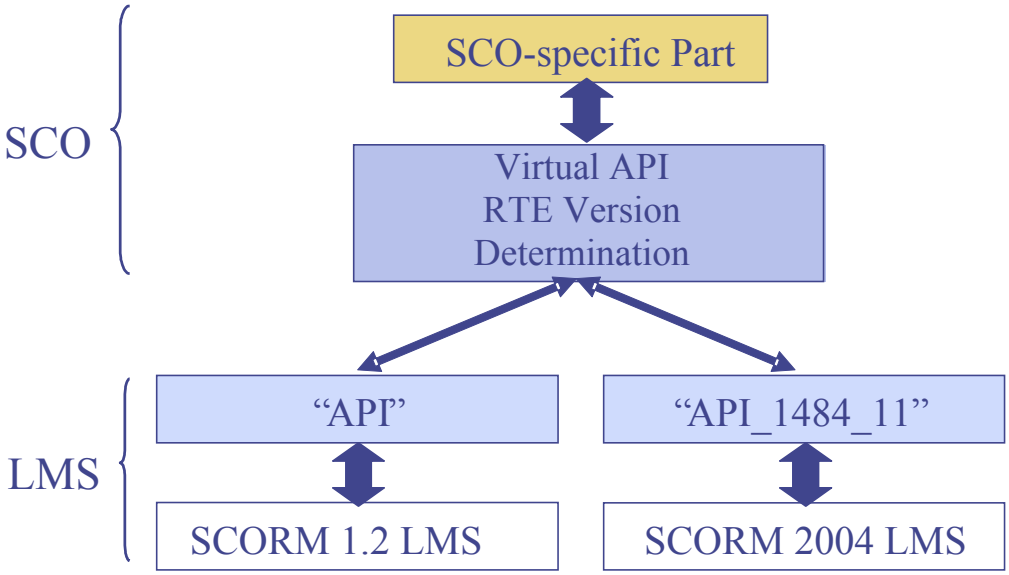


Figure 8.1 Implementing an SCO for both SCORM 1.2 and SCORM 2004

Index

About This Document

Acknowledgements

This document was a translation of a Japanese document authored by the following people:

Sections 1, 3, and 6 – 9:	Nakabayashi, Kiyoshi (NTT-Resonant Inc.)
Sections 2, 4, and 5	Miyauchi, Hiroshi (The Sanno Institute of Management) Ota, Mamoru (Enegate Co., Limited)

The publication of this document was sponsored by the Japanese Ministry of Economy and Industry for the purpose of promoting the SCORM 2004 standard.