SISO-GUIDE-009

Guide for Simulation Reference Markup Language – Primary Features

Version 1.0

9 March 2017

Prepared by Simulation Reference Markup Language Product Development Group

SISO Inc. Board of Directors
P.O. Box 781238
Orlando, FL 32878-1238, USA

Revision History

Version	Section	Date (MM/DD/YYYY)	Description
V1.0	All	08/14/2016	Circulation Package Version

Participants

At the time this product was submitted to the Standards Activity Committee (SAC) for approval, the Simulation Reference Markup Language Product Development Group (PDG) had the following membership and was assigned the following SAC Technical Activity Director:

Product Development Group

Robert Lutz (Chair)
Jane Bachman (Vice-Chair)
Curtis Blais (Secretary)

John Stevens (SAC Technical Activity Director)

Arjan Lemmers
David Ronnfeldt
Graham Shanks
Eugene Stoudenmire
Steven Weiss

The PDG would like to especially acknowledge those individuals that significantly contributed to the preparation of this product as follows:

PDG Drafting Group

Steven W. Reichenthal (Editor)
Paul Gustavson
Jeffrey Steinman

The PDG would like also to recognize the contributions of these other individuals, including current and former SISO members, who contributed to the development of this document.

Jeff Abbott
Rob Albright
Shelby Barrett
William Beavin
Paul Birkel
Jonathan Boan
Peter Bosch
Steve Boswell
Francis Bowers
Kevin Brandt
Pat Burgess
Cord Cardinal
Pablo Cases
Michel Charbonneau
Tram Chase
Guan Chaun
Bang Choon
Darrell Collins
Jeff Covelli
Daran Crush
Dannie Cutts
D Jacinto
Bradford Dillman
Arren Dorman
Arno Duvenhage
John Fay
Terrill Frantz
Masakazu Furuichi
Michael Gagliano
Juan Garcia
James Globe
Edward Gordon
Jean-Baptiste Guillerit
Paul Gustavson
Per Gustavsson
Kevin Heffner
Reinhard Herzog
Frank Hill
Jim Hollenbach
George Hughes
Theresa Hughes
Russell Hutt
Jae Jun Hwang
Bruce Jackson
John Jinkerson
Stephen Jones
Rajive Joshi
Anna Karphammer
Rosemarie Keener
Michel Keuning
Sam Knight
Jerry Kniphfer
Lee Krause
Lee Lacy
Jay Levenson
Bryan Linkous
Kok Seng Low
Van Lowe
Paul Lowe
Marianela Garcia Lozano
Franklin Lue
Farid Mamaghani
Peter Mandl

Lee Marden
Edward McCall
Michael McGarity
Robert McGraw
Wilhelm Medetz
Walter Miller
Francisco Montanes
Ronald More
John Morrison
Katherine L. Morse
Sean Murphy
Michael Myjak
Ekehard Neugebauer
Zac Ng
Tze Ng
John O'Reilly
Neal Oliver
William Oliver
Magnus Olsson
Shawn Parr
Ceri Pritchard
Greg Quedenfeld
Guillaume Radde
Steven Reichenthal
Andrew Robbie
Lawrence Root
Chris Rouget
Stefan Sandberg
Geoff Sauerborn
James Saultz
Christopher Scannell
Tom Schaefer
John Schloman
Terry Schmidt
Andrew J. Scholand
Steven Sheasby
Ray Shellman
Edward Shen
Ben Sisson
Jeffrey Steinman
Richard Strand
Duncan Suttles
Simon Taylor
Brian Tegowski
Joo Thio
Stephen Turner
Shon Vick
Bill Waite
Janet Wedgwood
John Welsh
John Woodyard
Jiang Wu
Zhai Yanlong
Mark Young
William Zimmerman

The following individuals comprised the ballot group for this product.

Ballot Group

Jane Bachman
Curtis Blais
Timothy Calderwood
Robert Lutz
Laurent Prignac
Marcy Stutzman

When the Standards Activity Committee approved this product on 9 February 2017, it had the following membership:

Standards Activity Committee

Katherine L. Morse (Chair)
Jean-Louis Igarza (Vice Chair)
Lance Marrou (Secretary)

Brad Dillman
Kevin Gupton
Paul Gustavson
Aerial Kreiner
Patrice Le Leydour
Chris McGroarty
Angus McLean
William Oates
John Stevens
Simone Youngblood

When the Executive Committee approved this product on 9 March 2017, it had the following membership:

Executive Committee

Michael O'Connor (Chair)
Bob Lutz (Vice Chair)
Jeff Abbott (Secretary)

Jim Coolahan
John Daly
John Diem
Lana McGlynn
Katherine L. Morse
Roy Scrudder
Robert Siegfried
Eric Whittington

Introduction

This guide provides a user guide to the primary features of the Simulation Reference Markup Language (SRML).

SRML is an XML-based data representation that provides the modeling and simulation community with:

- A common mechanism for loading and executing simulations.
- The ability to create an executable simulation model that can be shared between various devices.
- A way to improve model consistency between federated simulations.
- The ability to reuse simulations throughout the commercial, educational, and governmental communities, and across simulation domains.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

SISO-GUIDE-009, Guide for Simulation Reference Markup Language – Primary Features

1 OVERVIEW

This guide provides a user guide to the primary features of the Simulation Reference Markup Language (SRML). SRML is an Extensible Markup Language (XML)-based data representation that provides the Modeling and Simulation (M&S) community with:

- An interchange format to support the development of simulation models.
- A common mechanism for loading and executing simulations.
- The ability to create an executable simulation model that can be shared between various devices.
- A way to improve model consistency between federated simulations.
- The ability to reuse simulations throughout the commercial, educational, and governmental communities, and across simulation domains.

1.1 Scope

Three related documents comprise the SRML standard:

1. SISO-STD-009-00 [9] is the SRML specification that defines syntax and semantics of SRML markup for representing SRML models that can be executed in an SRML engine.
2. SISO-STD-009-01 [10] is the SRML engine specification that defines the required operation of an SRML engine.
3. SISO-GUIDE-009 (this document) is the SRML User Guide that provides a user–oriented guide to developing SRML simulations.

All of the information in this document is informative, rather than normative.

SRML provides markup to support the following capabilities:

1. Adding simulation structure to arbitrary XML documents, which may include markup from multiple schemas.
2. Associating behavioral scripts with arbitrary elements.
3. Associating groups of elements with classes in an XML-based compositional hierarchy.
4. Specifying classes of items in an item-class hierarchy.
5. Specifying classes of events in an event-class hierarchy.
6. Allowing the execution state of a simulation to be persisted and restored.
7. Describing quantities of individual items from a single element.
8. Establishing links between individual items.
9. Specifying global simulation control parameters.

The following are out of scope:

1. Providing markup in any specific domain outside the general-purpose simulation domain.
2. Specifying details regarding the performance of any simulation engine.

Although SRML provides features similar to those found in programming languages, such as class definitions, SRML is not a programming language. Instead, SRML is a markup language, like HyperText Markup Language (HTML), which provides features where scripts from programming languages may be included. It is a specification for structuring simulation content, which also offers the ability to include programming language scripts (e.g., JavaScript).

1.2 Purpose

The purpose of SRML is to provide a specification that enables simulations and other content to be served, received, and processed using web technologies. In many ways SRML is similar to how HTML provides a specification that enables text and other content to be structured, processed and viewed commonly using the

World Wide Web, which is hereon referred to as the web. MathML [22] is another example, where a specification enables mathematical notation, structure and content to be shared using web technologies.

The SRML specification defines a set of simulation-oriented constructs for inclusion with domain-specific XML content. This XML-based content can be used to define dynamic, web-based simulation models that encode the structure, behavior, and state of a simulation. General purpose XML tools can be used for building SRML models.

An SRML engine is software that is capable of executing SRML models.

Any application that consumes well-formed XML can augment that XML with SRML markup to express functional behavior associated with the document elements. The SRML formalism allows a data-driven behavior to be implemented for a wide spectrum of XML applications.

1.3 Objectives

The primary intention for this document is to present a manual for using SRML, suitable for introducing authors to SRML. It contains descriptive examples for new users.

SRML is designed according to the following goals:

1. Specify a flexible reference standard for representing simulations.
2. Provide enough expressive power to model most anything for the purposes of simulation.
3. Choose constructs that are simple, yet expressive, so that models can be developed and maintained using a text editor at minimum.
4. Specify a minimal set of predefined element names; maximize use of elements in user-defined schemas.
5. Support models defined in both single and multiple source files.
6. Maximize use of widely accepted standards such as XML, JavaScript, and allow other languages to be "plugged in".
7. Define semantics that are neutral and flexible, to enable simulation software vendors to implement SRML across multiple domains.
8. Select name constructs that minimize the ambiguity with reserved names found commonly in programming languages or in XML. For example the names: ‘class’, ‘event’, ‘entity’, ‘element’, and ‘attribute’ are avoided.
9. Provide a markup language similar to HTML, with individual element behavior and dynamic DOM functionality, yet targeted for simulations.
10. Provide object-oriented capabilities for modeling generalization and containment relationships, as well as multiplicity.
11. Maintain device and presentation independence.

1.4 Intended Audience

The primary audience for this guide is the M&S community; those who are interested in the modeling, interoperability, reusability, componentization, and composition of systems and simulations. Other communities with similar interests are encouraged to use this language in their domains.

1.5 Document Conventions

The key words *must*, *must not*, *required*, *shall*, *shall not*, *should*, *should not*, *recommended*, *may*, and *optional* in this specification are to be interpreted as described in SISO-ADM-005-2011 [4].

In this document, the words *define*, and *declare* refer to the use of SRML elements, whereas the word *specify* refers to the definition of the SRML elements themselves.

2 REFERENCES

2.1 SISO References

#	Document Number	Title	Date
1.	SISO-ADM-001-2014	Policy for Numbering of SISO Products	15 May 2014
2.	SISO-ADM-002-2011	SISO Policies and Procedures	11 Apr 2011
3.	SISO-ADM-003-2011	SISO Balloted Products Development and Support Process (BPDSP)	14 Nov 2011
4.	SISO-ADM-005-2011	Policy for the Style and Format of SISO Documents	13 Jun 2011
5.	SISO-PN-009-2014	Product Nomination for Simulation Reference Markup Language	13 Jan 2014
6.	SISO-REF-013-2005	Simulation Reference Markup Language Study Group Final Report	21 Apr 2005
7.	SISO-STD-003-2006	Standard for Base Object Model (BOM)	8 May 2006
8.	SISO-STD-003.1-2006	Guide for BOM Use and Implementation	8 May 2006
9.	SISO-STD-009-00	Standard for Simulation Reference Markup Language	9 March 2017
10.	SISO-STD-009-01	Standard for Simulation Reference Markup Language Engine Specification Level 0	9 March 2017

2.2 Other References

#	Document Number	Title	Date
11.	IEEE Std 1516.1™--2010	IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)--Federate Interface Specification	18 Aug 2010
12.	IETF RFC 3986	Uniform Resource Identifier (URI): Generic Syntax	2005
13.	W3C REC-DOM-Level-3-Core-20040407	Document Object Model (DOM) Level 3 Core Specification, Version 1.0	7 April 2004
14.	W3C REC-xml-20081126	Extensible Markup Language (XML) 1.0 (Fifth Edition)	26 Nov 2008
15.	W3C REC-xml-names-20060816	Namespaces in XML 1.0 (Second Edition)	16 Aug 2006
16.	W3C REC-xpath20-20070123	XML Path Language (XPath) 2.0	23 Jan 2007
17.	W3C REC-xslt-19991116	XSL Transformations (XSLT) Version 1.0	16 Nov 1999
18.	W3C xmlschema-0-20041028	XML Schema Part 0: Primer Second Edition	28 Oct 2004
19.	W3C xmlschema-1-20041028	XML Schema Part 1: Structures Second Edition	28 Oct t2004
20.	W3C xmlschema-2-20041028	XML Schema Part 2: Datatypes Second Edition	28 Oct 2004
21.	Scientific American 237, no. 3	Kay, Alan, "Microelectronics and the Personal Computer", Scientific American, pp. 230-244	Sep1977
22.	W3C REC-MathML2-20031021	Mathematical Markup Language (MathML) Version 2.0 (Second Edition)	21 Oct 2003
23.	W3C REC-owl2-features-20091027	OWL 2 Web Ontology Language Document Overview	27 Oct 2009
24.	W3C REC-rdf-concepts-20040210	Resource Description Framework (RDF): Concepts and Abstract Syntax	10 Feb 2004
25.	W3C REC-xhtml1-20020801	XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition), A Reformulation of HTML 4	1 Aug 2002

#	Document Number	Title	Date
		in XML 1.0	
26.	W3C REC-xml-events-20031014	XML Events, An Events Syntax for XML	14 Oct 2003

3 DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

3.1 Definitions

Term	Definition
Author	A person who develops an SRML model.
Behavior	A set of operations performed by an item as actions or reactions to events.
Class definitions	A set of item class and/or event class definitions.
Document order	The order in which elements occur in the XML representation of a document [16]—corresponds to a preorder traversal of DOM nodes.
Domain	A field of interest or study for which a simulation is constructed, such as transportation, logistics, communication, marketing, physics, etc. A domain includes a specific vocabulary, for example the vocabulary for a transportation domain includes vehicle, cargo, passenger, and route.
Event class	A description of the common characteristics of all item events having the same name.
Host	Any software that serves as the environment for an SRML engine.
Item class	A description of the common structure and behavior of all items to which it applies.
Item event	Information describing something that happens to an item at a point in time during the execution of simulation. Every item event has a name.
Item scope	A location that provides a boundary whereby the items it contains may be uniquely identified.
Item	An XML representation of the state of particular simulation object instance in terms of its attribute values at a point in time. Item also refers to an object instance in the run-time environment of an SRML engine.
Item-deriving element	Any XML element that an SRML engine uses to derive an item. Elements such as Item and Component are item-deriving elements, whereas Property, Script, Link, and EventSink are not.
Link	A property of an item that references another item. An item may have links to multiple other items.
Local identifier	A name used once within an item scope to identify an item.
Location	Refers to an item's role when it serves as a place where other items reside. A location is derived from a DOM "parent node". An item may be at only one location at a time, but may be transitively at many locations.
Model content	A portion of an SRML model that describes the items and their links at a point in time. Model content may be specified using constructs from the SRML namespace or in conjunction with markup from other namespaces.
Namespace qualification	The explicit association between a name and its namespace, by reference to the namespace's URI.
Namespace	A collection of element, attribute, data type, and other names to be used in XML documents, and identified by a Uniform Resource Identifier (URI) [12].
Owner (owning item)	An item that includes a particular occurrence of a property, script, link, or event sink.
Property	A named feature, or characteristic of an item.

Property-deriving attribute	Any XML attribute that a simulator uses to derive a property.
Reference element	One of several elements in the SRML namespace that is intended to be used as a template for authors when defining the elements in their own namespaces.
Script	Any text that describes executable instructions that an SRML engine uses to execute the behavior of an item.
Simulation structure	Refers to the organization or augmentation of model content for the purpose of running simulations.
SRML engine	Any software that is capable of loading and executing SRML models.
SRML model	An XML representation of a simulation that uses constructs from the SRML namespace mixed with model content from other namespaces to describe the structure and behavior of the items to be simulated.
SRML simulation	The execution of one or more SRML models within an SRML engine.
Target	An item referenced by a link.

3.2 Acronyms and Abbreviations

Acronym or Abbreviation	Meaning
BOM	Base Object Model
DOM	Document Object Model
EXCOM	Executive Committee
HLA	High-Level Architecture
HTML	Hyper Text Markup Language
M&S	Modeling & Simulation
MathML	Math Markup Language
PDG	Product Development Group
SAC	Standards Activity Committee
SISO	Simulation Interoperability Standards Organization
SRML	Simulation Reference Markup Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XSD	XML Schema Definition

4 BACKGROUND

Simulation is a process that attempts to predict aspects of the behavior of a system by creating an approximate (or mathematical) model of it. Computers provide an ideal environment for building and executing simulations. Simulations have many uses, including the following:

- Businesses use simulations to develop and optimize processes, to help lower risks associated with critical decisions.
- Engineers use simulations to prototype and test designs.
- Researchers use simulations to validate hypotheses.

In a broader sense, computing can be viewed as simulation [21] since computer programs resemble the domains to which they are applied. Thus, even though the traditional view of simulation is expressed in this document, SRML may be used to implement broad computing solutions.

Generally, the simulation process follows a three-step process: Model, Simulate, and Evaluate, as shown in Figure 1. This process involves several roles: modeler, domain expert, recipient, decision maker. Modelers and domain experts (or authors) build models using a tool belonging to a "simulation development environment". This tool generates and stores a representation of the model that is fed into a simulation engine that manages objects and events in a run-time environment consisting of support software that provides mathematics, statistics, random numbers, event management, and other analytical capabilities. The simulation engine typically provides some kind of feedback about the execution progress, as running a simulation may take more than a few seconds. Once the simulation has completed, the participating modelers, domain experts, and decision makers evaluate the results and feed those results into the construction of a new model. Thus, simulation usually follows an ongoing iterative methodology.

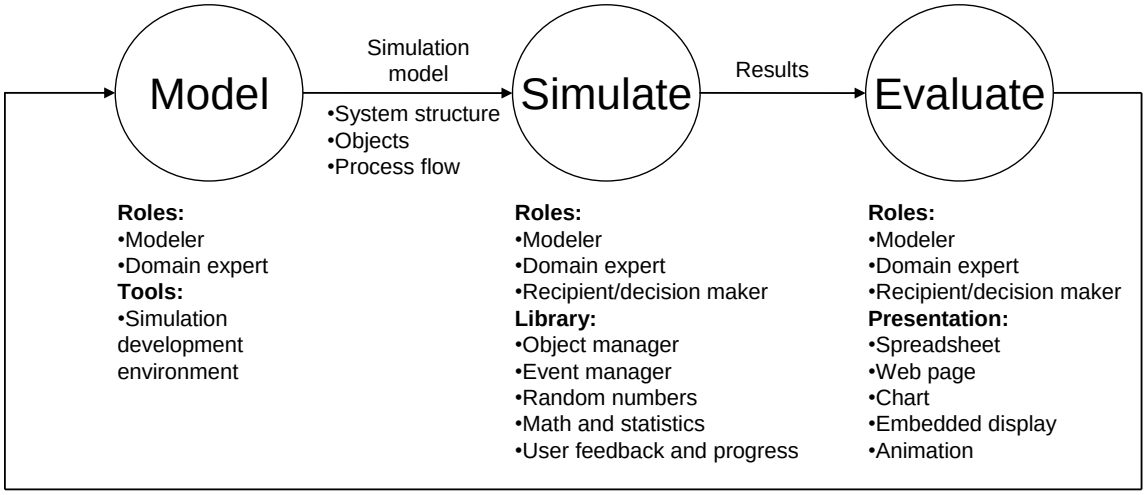

Figure 1: Simulation Development Process

A modeler who uses a general-purpose simulation development environment of any complexity is doing a form of programming, whether that programming is text-based or graphics-based. Therefore, simulation and programming are driven by similar factors such as cost, ease of programmability, portability, usability, reusability, and adherence to standards. Aside from these commonalities, simulations pose challenges of their own. For example, simulations typically need to represent large numbers of interrelated objects, and need to manipulate them at high speeds. Thus, unlike a traditional business program, a simulation may require lots of computing speed and memory. Due to the iterative nature of simulation, participants typically need a way to easily repeat the execution of a simulation with varying random number seeds, to vary one or more parameters, and to be able to combine the results.

Simulation users also need the ability to watch, freeze, reverse, snapshot, monitor, alter, and produce results from a simulation. Thus, user interaction and presentation become a major part of the process. Information presented from a simulation can take many forms, including spreadsheets, charts, web pages, and animations. Supporting multiple presentation representations can pose challenges that are beyond the scope of the actual problem to be solved by the simulation.

Additional challenges arise in simulations because they can have many characteristics, including:

- Discrete event (Monte Carlo methods, random events, probabilities) vs. continuous
- Distributed vs. individual
- Parallel vs. serial processing
- Single run vs. continuous execution
- Interactive vs. batch
- General-purpose vs. specific-focus
- Live vs. Virtual vs. Constructive
- Single resolution vs. multiple resolution

Simulation requires an investment in tools. Tools belonging to a simulation development environment range from spreadsheets to programming languages to full commercial simulation packages. Each has tradeoffs. Spreadsheets, for example, offer a low purchase cost, but can suffer in performance and flexibility. Traditional programming languages provide maximum performance and flexibility but can require more time and skill in implementing the model to be represented, or in re-inventing the simulation engine necessary to execute the simulation models. Even with a simulation engine library, programming language solutions usually do not provide a natural means for describing hundreds or thousands of interconnected objects of various types. General-purpose commercial simulation tools provide sophisticated built-in capabilities combined with graphical development environments that can save time, however, they often come with a high price tag, produce models that are difficult to maintain over time, and typically don t operate well with other tools. Fortunately, the advancements of technologies such as XML have made it possible to define a specification that allows for a greater level of reuse, integration, and sharing of executable models.

4.1 XML Motivation

With the widespread use of the Internet, XML was developed as a standard for structured transmission and representation of arbitrary data. As such, XML has several characteristics that make it particularly suitable and beneficial for representing simulations:

- XML has enough expressiveness to describe data objects at various levels of complexity.
- Through the use of schemas, XML provides structural document validation and data-typing.
- XML can be easily transported in a human readable form to various places.
- XML can easily describe hierarchical structures and linked networks.
- Once loaded, XML data can be queried in a fashion similar to a database.
- XML editing software is relatively inexpensive to obtain.
- Standard software exists for reading and writing XML, e.g. the Document Object Model (DOM).
- Scripting can be embedded within XML to add behavior.
- XML tools can support a simulation run-time environment.

4.2 XML Foundation

This section provides a brief introduction to XML for understanding this specification. References are provided for further reading.

XML [14] provides constructs that may be used to describe hierarchically related discrete data. XML consists of “elements” that are specified using tags (words bracketed by '<' and '>'). Elements can include attributes (described by name-value pairs), and can also contain embedded elements and text content.

Listing 1: Generic XML template:

```
<?xml version="1.0"?>
<elementname1 attributename1="value1" attributename2="value2"
    xmlns:prefix1="someuri">
    textcontent1
    <prefix1:embedded-element1 xmlns:prefix1="uri-of-namespace1">
        textcontent2
    </prefix1:embedded-element1>
</elementname1>
```

Listing 2: Populated XML template:

```
<?xml version="1.0"?>
<workers capacity="22" overthreshold="4" xmlns:a="urn:workermodels">
    <a:worker Name="workerA" busy="False">
        BADGE7
    </a:worker>
    <a:worker Name="workerB" busy="False"/>
</workers>
```

Well-formed XML adheres to basic rules that apply to all XML documents. For example, every start tag <x> must either have a matching end tag </x> or otherwise be represented as an empty tag <x/>, without content.

When a tag contains a colon, the prefix (before the colon) refers to a namespace, and the suffix (after the colon) is a local name defined within the namespace [15]. This provides a means for identifying markup that has been mixed from various sources. The uniform resource identifier (URI) of the namespace is declared by the value of an attribute that is named using the “xmlns” prefix and an optional suffix declaring the namespace name. Listing 1 illustrates a generic template for an XML file, and Listing 2 shows XML with actual content.

Rules governing the allowable element and attribute names within a namespace may be described using an XML Schema [18,19,20]. Such rules can be used to validate particular XML content with software.

The XML Document Object Model (DOM) [13] is an Application Programming Interface (API) that enables programming languages and scripts to load, validate, navigate, modify, and save XML structures with code. DOM software implements elements and attributes with linked Node objects. For example, the previous XML template in Listing 1 could be represented in the DOM as shown in Figure 2.

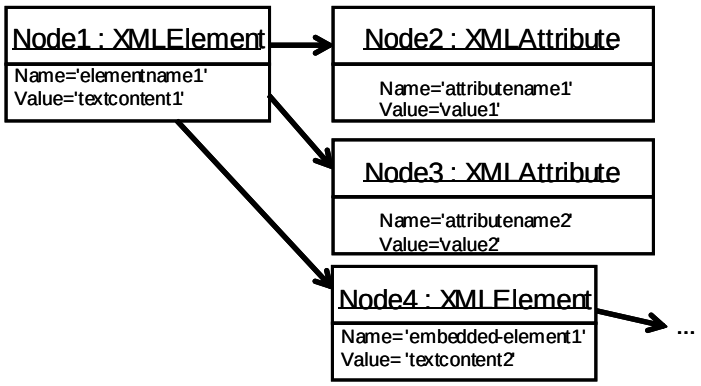

Figure 2: Document Object Model (DOM) Example

4.3 Related XML Works

Use of plain XML does not imply semantic meaning, presentation, or behavior of data. Rather, higher-level specifications are intended for those purposes. RDF/OWL [23,24] provides for the expression of semantic meaning. HTML/XHTML [25] provides for user presentation coupled with behavior. SRML provides for simulation behavior. Typically, behavior is programmed into software that is specifically designed to process a particular schema.

Display-oriented markup languages like HTML and others that render in 3D may provide scripting as a means for modeler-defined behavior, but the tight coupling of presentation with behavior can make it awkward to separate model from behavior when an alternative form of presentation (or view) is desired. The aim of SRML is to provide a language similar to HTML in structure, except that SRML eliminates the presentation capability and enhances the behavioral and dynamic DOM functionality for purposes of simulation, thus allowing alternate presentation layers to share the same fundamental simulation model representation.

Other specifications in the field of M&S use XML as a data interchange format, most notably are the High-Level-Architecture (HLA) [11] and Base Object Model (BOM) [7,8] standards. While these standards provide markup that facilitates the conceptualization and information interchange among member applications participating in a distributed simulation environment, they do not provide features for describing executable behavior or the individual states of simulation object instances. Thus, SRML used in conjunction with these standards provides a complete set of standards for fully describing the structure and behavior of both individual and distributed simulations.

4.4 Relationship Between SRML and XML

Architecturally, SRML may be thought of as a layer above XML. SRML defines a domain that overlays the XML domain so that arbitrary XML content may be treated as a simulation model by default or through augmentation.

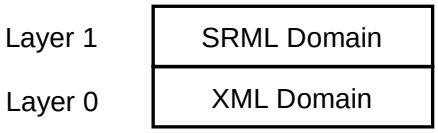

Figure 3: SRML and XML Domain Relationships

The XML domain specifies: documents, elements, attributes, child elements, parent elements, and id references. Correspondingly, SRML specifies: simulations, items, properties, sub-items, locations, and links. An SRML Simulator derives simulations from documents, items from elements, properties from attributes, sub-items from child elements, and locations from parent elements.

Besides the domain correspondence, SRML further specifies item classes, behaviors, event classes, event sinks, events, and parameters. Although an XML Events specification does exist as a standard [26], its capabilities are more user-interaction centric and less simulation centric than those in SRML.

Table 1 maps the relationships between the SRML and XML domains, and indicates which relationships are derived from XML content.

Table 1: SRML to XML Mapping

SRML Domain	Derived	XML Domain
Simulation	Yes	Document
Item	Yes	Element
Property	Yes	Attribute; “attribute” definition in XML Schema
Sub-Item	Yes	Child Element
Items	Yes	childNodes in the DOM
Location	Yes	Parent Element, parentNode in the DOM
Links, Link	Yes	IDREF(s) and XPath, XLink
ItemClass	No	“element” definition in an XML Schema
Script (Behavior)	No	-
EventClass	No	-
EventSink	No	-
Events	No	-
ItemEvent	No	-

SRML is intended to augment the element and attribute definitions developed by authors. This encourages modelers to use or define other domain-specific XML schemas while developing model content.

4.5 Relationship Between SRML and XML Schema

SRML item classes and property definitions are similar to XML Schema’s element and attribute definitions in that they all provide “type information”. The complex type definitions in XML Schema are used to describe the structure of valid content, whereas the item class structures in SRML are used to augment the structure of content. SRML does not provide features to validate document structure, nor does XML Schema provide features to define simulation behavior.

The two may be used together. Simple type definitions made using XML Schema may be used in SRML property type definitions. Likewise, XML Schema’s complex definitions may be used to validate SRML item class structures. Conversely, schema authors may use SRML-defined data types and elements when designing their schemas.

4.6 Software Used with SRML

The following software is used with SRML:

- Model editor
- SRML model
- SRML engine
- A host

A model editor is any software that can assist in the construction or maintenance of SRML models. Such a tool may be a plain text editor, an XML editing tool, a custom SRML editing tool, or software that generates SRML from another representation.

An SRML model is an XML description of a simulation that conforms to the SRML language specification. The goal of the SRML language specification is to provide guidelines for building models that can be run in any SRML engine. SRML was designed with minimal requirements, allowing most any XML to be loaded into an SRML engine.

An SRML engine is software that is capable of loading an SRML model and executing that model in accordance with the SRML engine specification. The goal of the SRML engine specification is to provide guidelines for building and using SRML engines to run any SRML model. The choice of a particular SRML engine implementation (vendor product) depends on application-level requirements, and the constraints of a host.

A host is software that is capable of instantiating and communicating with SRML engine to run a simulation. Examples of hosts are web browsers, macro environments such as those found in office applications, and embedded software.

Other software may be needed for a particular SRML model to execute, such as plug-in components that are compatible with the SRML engine. For example, a specific plug-in language processor may be necessary for executing a particular type of script, such as a plug-in JavaScript or Python compiler. Other types of plug-ins may provide specialized communications or routines for advanced math functions.

This user guide is primarily about the principles necessary to build SRML models, and to execute those models in an SRML engine. Editors, hosts, and other software will have their own user guides.

Figure 4 shows how the software fits together.

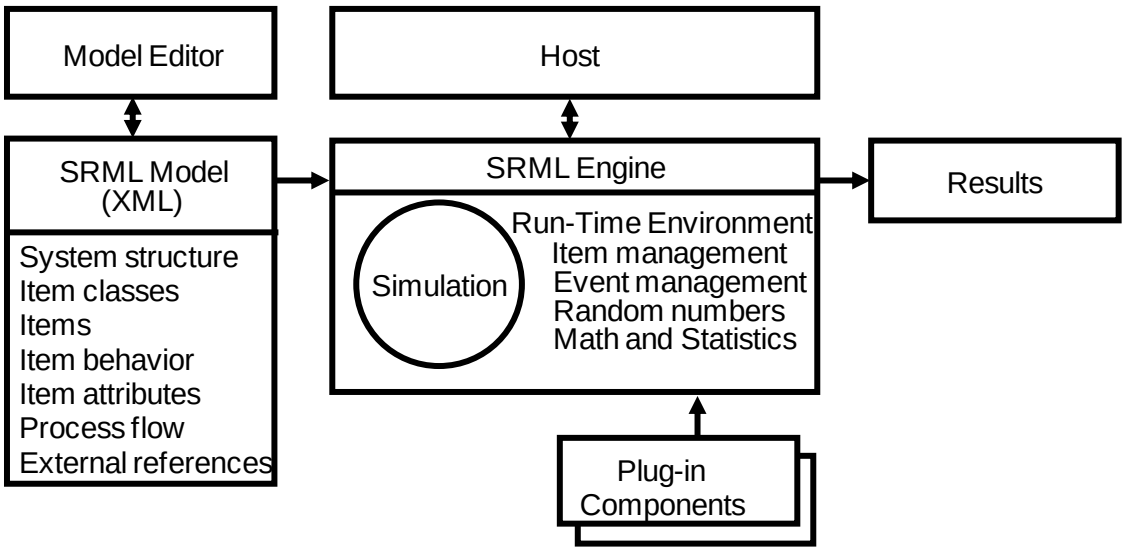

Figure 4: Simulation Context

4.7 SRML Model Fundamentals

An SRML model is an XML representation of a simulation that uses any combination of constructs from the SRML namespace mixed with constructs from other namespaces to describe the structure, behavior and state of the items to be simulated.

The things described by an SRML model are referred to as items. An item can represent a physical thing, such as a piece of equipment, or a person, or an entire system of other items. An item may also represent an abstract thing such as a process or a step in a process. Items can have properties. For example, a piece of equipment might have a 'serial number' property, and a process might have a 'duration' property. SRML provides a natural way to express items, their properties, and complex item relationships, using the grammar of XML with a small set of pre-defined SRML elements and attributes that have specific meanings and rules. Listing 3 shows three simple, yet complete, SRML models:

Listing 3: Three simple models

```
<!-- first model -->
<Simulation/>

<!-- second model -->
<Counter Value='47'/>

<!-- third model -->
<Process Name='Create' Duration='46'/>
```

The first model defines a single Simulation item that contains no other items—other items could be added after the model is loaded. The second model defines a counter item that has an initial value. The third model defines a process item with a name and duration. Each of these models may be loaded successfully into an SRML engine, either individually or as components of larger models. Amongst the three, the model containing the Simulation item is the only one that uses a pre-defined SRML element that has pre-defined behavior. The other elements are ad hoc—without additional definition they have only intrinsic behavior.
Items and properties correspond to XML elements and attributes. When an SRML engine loads an SRML model, items are derived from the elements, and properties are derived from the attributes. With the exception of a few pre-defined element names, such as Simulation, SRML uses the element and attribute names that appear in a model when creating items. This makes it convenient to create a domain-specific XML schema to validate the structure of a system and to provide data types for the attributes. A domain-specific schema works in conjunction with the SRML Schema when using XML namespaces.

5 SRML ENGINE FUNDAMENTALS

One must obtain or otherwise develop the software of an engine before they can use SRML for executing models. This guide assumes that the reader has an SRML engine. The standard capabilities of an SRML engine implementation are specified in the SRML Engine Specification. A primary capability of an SRML engine is as a component for managing Simulation objects.

5.1 Instantiating a Simulation

Before a model can be executed with a SRML engine, a Simulation object must be instantiated within a host environment. A standard factory method of an SRML engine is CreateSimulation, which returns a new object that has a Simulation interface. Listing 4 is an example of the assignment of a new Simulation object to a variable named “Simulation” by using the CreateSimulation method using a programming language such as JavaScript. In the listing, the identifier “SR_Simulator” refers to the name of a particular SRML engine implementation. In practice, this name may be replaced by the specific product’s engine identifier.

Listing 4: Instantiating a Simulation object using Create Simulation

```
Simulation = SR_Simulator.CreateSimulation()
```

Additionally, each host environment may bring different ways to instantiate Simulation objects. Refer to Listing 5 for an example that describes the instantiation of a Simulation object using HTML.

Listing 5: Instantiating a Simulation object using an HTML "object" element

```
<html>
<head>
<object id="Simulation" type="application/srml"></object>
</head>
</html>
```

It's important to recognize that the combination of host and engine will determine the means by which an SRML engine will be instantiated. All of the examples in this guide were developed using a web browser as a host and a product called "SR_Simulator".

5.2 Loading an SRML Model

Models may be loaded by an SRML engine as strings of characters with the LoadXML method of a Simulation object, as shown in Listing 6. Another means for loading a model is with the Load method, as shown in Listing 7.

Listing 6: Using LoadXML to load a model

```
Simulation.LoadXML("<Counter Value='47'/>")
```

Listing 7: Using Load to load a model

```
// Load from a relative URL.
Simulation.Load("Counter01.xml")

// Load from a absolute URL. (Note: example URL may not actually exist.)
Simulation.Load("https://sisostds.org/srml/samples/Counter01.xml")
```

5.3 Running a Simulation

Once loaded, a model becomes a simulation that contains items as objects which can be evaluated and manipulated by software. A Simulation object can also execute the model as individual steps (using DoNextEvent), or as a single run (using the Run method). In all cases, a Simulation object makes notifications to the host about changes that occur within the simulation.

The example code in Listing 8 shows the running of a model called "test.xml" from an HTML page in a web browser host. When any event occurs in the simulation it will added to the end of the text on the page.

Listing 8: Running a simulation in a web browser

```
<html>
<head>
<title>Simulation Demonstration</title>
<object id="Simulation" type="application/srml"></object>
<script type="text/javascript">
function RunSimulation()
  {
  Simulation.CurrentTime = new Date("1/1/2001").getVarDate ();
  Simulation.EndTime = new Date("1/2/2001").getVarDate ();
  Simulation.Load("test.xml");
  Simulation.RandomizeSeed();
  Simulation.Run();
  }
</script>
<script for="Simulation" event="EventOccurred(ev)" type="text/javascript">
  document.body.appendChild(document.createElement("<P>"));
  var message = ev.Time + " " + ev.Target + ev.Target.ItemID + " " + ev.Name
  document.body.appendChild(document.createTextNode(m));
</script>
</head>
<body>
<button onclick="RunSimulation()" type="text/javascript">Run Simulation</button>
</body>
</html>
```

6 INTRINSIC ITEM CAPABILITIES

Some common properties and methods are built into all items. For example, every item provides the following fundamental capabilities:

1. Navigation to its sub-items (inner items) and locations (outer items)
2. Finding other items
3. Linking to other items
4. Providing general item information such as a count of sub-items.

The model in Listing 9 will be used for demonstrating these fundamental capabilities.

Listing 9: Market example

```
<Market>
  <Aisle Name="Freezer" Number="17">
    <IceCream UPC="4154806185" Title="Mint Chocolate Chip" />
  </Aisle>
  <Aisle Name="Cookies and Crackers" Number="4">
    <Crackers UPC="077544132009" Title="Sesame Crackers"/>
  </Aisle>
  <Checkout Number="15"/>
</Market>
```

After the model in Listing 9 is loaded into an SRML engine, it may be evaluated in host using code, such as with the JavaScript code shown in Listing 10.

Listing 10: Evaluating the Market

```
// Find the market inside the simulation.
market = simulation.FindItem("Market")

// Find the freezer inside the market.
freezer = market.FindItem("Aisle[@Name='Freezer']")

// Obtain the number of items in the freezer.
n = freezer.Items.Count

// Reference the first item in the freezer.
IceCream = freezer.Items(0)

// Get the item where the freezer is located.
X = freezer.location
```

Note: A complete list of intrinsic item properties and methods may be found in the SRML engine specification.

6.1 Finding Items

To dynamically find an item (or items) relative to a particular item, the intrinsic FindItem and FindItems methods will return a collection of items using XPath expression. For example, the code in Listing 11 will set a freezer variable to the first item found in a location that has its name set to Freezer.

Listing 11: Using FindItem and FindItems

```
// Find the freezer inside the market.
freezer = market.FindItem("Aisle[@Name='Freezer']")

// Collect all of the aisles.
aisles = market.FindItems("Aisle")
```

6.2 Intrinsic Item Identification

When an SRML engine initially creates an item object, it automatically assigns a unique numeric value to its 'ItemID' property. This property may be used to uniquely identify the item see Listing 12.

Listing 12: Accessing ItemID

```
// Get the ItemID of the freezer.
freezerItemID = freezer.ItemID
```

6.3 DOM Access

Each item has a Node property that returns its corresponding DOM node. This provides direct access to the underlying DOM architecture and its capabilities. In Listing 13, the Node is used to retrieve the ItemID attribute value.

Listing 13: Accessing the DOM through an item

```
// Get the ItemID of the freezer by using the DOM node.
freezerItemID = freezer.Node.getAttribute("ItemID")
```

6.4 Sub-items

Another intrinsic property of each item is its Items collection, which provides access to its sub-items—the items that are directly located within an item. This collection also has a Count property that returns the number of sub-items. See Listing 14.

Listing 14: Counting and accessing sub-items

```
// Obtain the number of direct sub-items of the market.
n = market.Items.Count // returns 2 (aisles)

// Address the first aisle.
aisle = market.Items(0)
```

6.5 Item Locations

Every item has a Location property that refers to its current location in the simulation item hierarchy. An item's location usually refers to another item, but sometimes it can be null, which means that the item either does not exist in the model or is at the top of the location hierarchy. The code in Listing 15 references an item’s location for later use.

Listing 15: Referencing an item’s location

```
// Reference the item where the freezer is located.
X = freezer.location
```

7 LATENT ITEM AND ENGINE CAPABILITIES

The SRML Language Specification describes “special data types” that constrain indicate the unique role of a property. When using the XML attributes that are based on these data types in a particular model, particular operations are activated, such as the production a specified quantity of items from a single element, or an item scope placed at a particular location.

Item scope and item identification

Item scope refers to a location (which is itself an item) that provides a boundary whereby the items it contains may be uniquely identified with a name. The default item scope encompasses the outermost item in the simulation. Within the default item scope any item can be identified by setting its ID property. An ID must be unique within the entire model since it always applies to the default item scope. Using global identifiers may be too limiting, as a particular item may need to be identified relative to a fragment of a model. Therefore, SRML provides the ability to define arbitrary item scopes where local identifiers can be unique only to that scope.

Any item can be used as an item scope at a particular location in a model by setting its ItemScope property. To make a local identifier the LocalID attribute is used. In fact, since every model has a default item scope,

LocalID could be used instead of ID. An item scope can be open or closed. An open scope includes the local identifiers from its outer scope and a closed scope does not.

The model in Listing 16 declares two item scopes: B1 and B2, which both contain an F1. Refer to Listing 17 as each F1 is accessed in code through its scope.

Listing 16: Declaring item scopes

```
<Simulation>
  <Building ItemScope="B1" ScopeClosed="true">
    <Floor LocalID="F1" Capacity="100"/>
  </Building>
  <Building ItemScope="B2" ScopeClosed="true”>
    <Floor LocalID="F1" Capacity="200"/>
  </Building>
</Simulation>
```

Listing 17: Accessing items through item scopes

```
Simulation.Item("B1").F1.Capacity // 100
Simulation.Item("B2").F1.Capacity // 200
```

7.1 Items with Fixed Locations

Some items, such as geographical locations, may have their locations fixed for an entire simulation. When these items have unique names within their containing locations, such as California within United States, they can be made directly accessible by specifying a LocationFixed='True' attribute value and a Name. Refer to the example in Listing 18 for the declaration a hierarchical set of fixed locations, and Listing 19 for code that references an item at one of those locations. In the code, the first Truck in Sacramento is accessed through the fixed locations in the World.

Listing 18: Declaring fixed locations

```
<Location ID='World'>
  <Location Name='USA' LocationFixed='True'>
    <Location Name='California' LocationFixed='True'>
      <Location Name='Sacramento' LocationFixed='True'>
        <Vehicle Name='Truck' Quantity='100' />
      </Location>
    </Location>
  </Location>
</Location>
```

Listing 19: Referencing an item at a fixed location

```
X = Simulation.Item("World").USA.California.Sacramento.Items(0)
```

7.2 Item Quantities

A situation may arise when a quantity of a particular item or structure needs to occur. Setting a positive integer value to the Quantity attribute instructs an SRML engine to duplicate the element and its contents. For example, Listing 20 uses quantities to define a total of 240 eggs in a total of 20 egg cartons in two refrigerators. The count is obtained by the code in Listing 21.

Listing 20: Declaring item quantities

```
<Simulation>
  <Refrigerator Quantity='2'>
    <EggCarton Quantity='10'>
      <Egg Quantity='12' />
    </EggCarton>
  </Refrigerator>
</Simulation>
```

Listing 21: Counting the eggs in the refrigerators

```
nEggs = Simulation.FindItems("//Egg").Count
```

7.3 Item Links

Non-hierarchical connection can be made between items by using links which define item properties that refer to arbitrary items. The SRML Link element has a Name to identify the link, and an XPath expression to identify the target item. See Listing 22 for an example of declaring a link from a vehicle to a dealer, and Listing 23 for code that accesses the dealer through the vehicle to retrieve the dealer's volume.

Listing 22: Declaring links

```
<Simulation>
  <Vehicle ID="V123">
    <Link Name="Dealer" Target="//*[@DealerID='333']"/>
  </Vehicle>
  <Dealer DealerID="333" Volume="1000" />
</Simulation>
```

Listing 23: Accessing an item through a link

```
X = Simulation.Item("V123").Dealer.Volume
```

7.4 Changing an Item's Location

The static structure of a model represents interconnected items at a point of time. As a simulation progresses, an item's location may change. This may be accomplished by simply assigning its location property to a different item. Another way to set the item's location is to use the SetItemLocation method of a Simulation object. For examples of these two options, the code in Listing 24 changes IceCream item's location to the Checkout station numbered 15.

Listing 24: Setting an item's location

```
Checkout15 = market.FindItem("Checkout[@Number='15']")

IceCream = market.FindItem("//Aisle[@Name='Freezer']/IceCream")

IceCream.Location = Checkout15

Simulation.SetItemLocation(IceCream, Checkout15)
```

Whenever an item's location is changed, the simulator will notify its host by raising the ItemLocationChanged notification, and passing the item along with its previous location.

8 SCRIPT BEHAVIOR

One or more Script child elements containing programming or scripting language may be placed directly inside an item's element to define custom behavior. Use the Script element's Type attribute to define which language is used. Depending on the particular SRML engine, a language may be built-in or it may need to be installed separately. An engine implementation may also allow the mixture of separate scripts from different languages within the same item. As a simulation runs, the scripts operate directly on the underlying DOM nodes as they alter item properties and locations.

Refer to the example in Listing 25. In the example, the Blender item is given a unique system-wide identifier (an ID property has global semantics) and a Running property with an initial value of '1'. The script directly accesses DOM attributes to obtain properties. Note that the item's behavior comes by way of author-defined functions, such as 'turnOn', and 'turnOff'. Because the script defines Blender01 as having a global identifier, the following code examples may be used anywhere in the model. Listing 26 executes the behavior that was defined by the script.

Listing 25: Blender with script behavior

```
<Blender ID='Blender01' Running='1' Speed='100'>
  <Script Type='text/javascript'>
    function turnOn()
      {
      Running=1;
      }
    function turnOff()
      {
      Running=0;
      }
    function OnSpeed(ev)
      {
      Speed=ev.Arguments(0);
      ev.Cancel = true;
      }
  </Script>
</Blender>
```

Listing 26: Blender with script behavior

```
Blender01.turnOn();
X=Blender01.Running;
Blender01.turnOff();
```

All items have an intrinsic 'Self' property that refers to the particular item instance in which a script is running.

9 EVENT MANAGEMENT

Some items correspond to physical things, which include people, the weather, the ocean, the planets, mechanical devices and electronic circuits. Items such as these may send and receive stimuli through the dispatching of events and the implementation of event sinks.

9.1 Event Dispatch Procedures

The SRML runtime environment provides procedures for dispatching events in the following ways: synchronously, asynchronously and scheduled.

Each of the dispatch procedures takes a target object, an event identifier, and a variable number of parameters. Events can be directed to an individual target object or to all objects by specifying a null target. Target objects are never required to handle events.

9.2 Event Formats

For an item to receive an event, it must implement a method to be invoked when the event has been triggered. The method may function using one of two formats. The first format, the “method” format, has a signature that consists of the method name followed by the parameters. The second format, the “item event” format, has a signature that consists of the method name and one parameter, an ItemEvent object, which contains the pertinent information about the event such as the sender and parameters. It’s important that the dispatching and receiving event formats are compatible. Therefore, the dispatching procedures exist in both formats:

Synchronous:	SendEvent, SendItemEvent
Asynchronous:	PostEvent, PostItemEvent
Scheduled:	ScheduleEvent, ScheduleItemEvent

9.3 Synchronous Events

Synchronous events are dispatched and received immediately by a target object, and the sending procedure cannot proceed until the event handlers have all completed—this is sometimes called blocking. The SendEvent method may be used to dispatch a synchronous event and retrieve a result.

An event dispatched like that in Listing 27 will invoke a procedure (if it exists) with a signature like this: “`function turnOn()`.”

Listing 27: Sending a synchronous event using SendEvent

```
result = SendEvent(Blender01,'turnOn')
```

Note: Over use of synchronous events may yield negative consequences on the scalability of a model in time and space.

9.4 Asynchronous Events

Asynchronous events are dispatched to an event queue, rather than being processed immediately. The sending procedure does not wait until the target receives the event. Asynchronous events are received by the target object(s) in order of priority before any scheduled events. The PostEvent method may be used to send an asynchronous event. It returns a handle which may be used to cancel otherwise obtain information about the posted event. A handle is returned from this procedure to allow the event to be retracted.
Refer to Listing 28 for an example of posting the TurnOn event to the blender.

Listing 28: Posting an asynchronous event using PostEvent

```
handle = PostEvent(Blender01,'turnOn')
```

9.5 Scheduled Events

Scheduled events are dispatched to an event queue with a date-time value that designates when the event should be processed. The events are received by the target object(s) in order of priority after any and all asynchronous events have been processed from the queue.

Refer to Listing 29 for an example of scheduling the TurnOn event to be received by the blender at a particular date and time.

Listing 29: Scheduling an event using ScheduleEvent

```
handle = ScheduleEvent(Blender01,'turnOn', dtmNextFriday)
```

9.6 Cascading Events

Synchronous, Asynchronous, and Scheduled events may also be dispatched so that they cascade to items that are hierarchically related to the target item. This means that after the item is received by the specified target item, the same event will also be sent up or down the item hierarchy—cascading upward, or cascading downward. Cascading upward means that after the target item receives the event, the location receives the event, and so on to the root of the simulation. Cascading downward means that after the item receives the event, the first child receives the event, and onward to all the children in document order (depth first).
An event sent to the blender that cascades down is dispatched as shown in Listing 30. It will invoke a procedure with a procedure called OnSpeed signature like the one shown in Listing 25, which sets the speed value and cancels the cascade operation.

Listing 30: Cascading downward

```
handle = PostItemEvent(Blender01, 'OnSpeed', srmlCascadeDown, 50)
```

9.7 Retracting Events

A particular scheduled or posted event may be cancelled prior to its occurrence by calling RetractEvent and passing the event identifier. Use an item's ItemEvents property to identify the events that are scheduled to be received by a particular target item. See Listing 31.

Listing 31: Retracting an event

```
RetractEvent(handle)
```

9.8 Event Sinks

When an item event is directed at a specific target item, that item will receive the event. However, some item events may be directed to any target that subscribes to the event. Those events are dispatched to null targets. Subscribing items use event sinks to subscribe to item events.

The example in Listing 32 defines a model with a Receiver1 item that declares an event sink that will respond to a Signal event, and will assign the event's argument value to its LastSignal property. Listing 33 contains code that posts an event to trigger the event sink.

Listing 32: Declaring an event sink

```
<Receiver LocalID="Receiver1" LockComplete="No" LastSignal="">
  <EventSink Name="TX"
     EventClasses="Signal" EventMethod="TX_SignalReceived"
     ItemClasses="SignalGenerator" WithPropertiesChanged="Frequency"
     PropertyMethod="TX_PropertyChanged" />
  <Script Type="text/javascript">
    <![CDATA[
    function TX_SignalReceived(e)
      {
      LastSignal = e.Arguments(0)
      }
    function TX_PropertyChanged(objTransmitter, PropertyName, NewValue)
      {
      LockComplete = "Yes"
      }
    ]]>
  </Script>
</Receiver>
```

Listing 33: Posting an item event to trigger event sinks

```
handle = PostItemEvent(null, 'Signal', srmlCascadeNone, 100)
```

9.9 Event Classes

Events can be represented simply by a name sent as a string parameter in an event generation procedure such as ScheduleEvent. However, just as items may be generalized into item classes, events may also be generalized into event classes. This allows events of the same name to share a common definition. For example, they might share the same priority and parameters. An event class defines static characteristics that apply to all item events of the same name.

In Listing 34, whenever turnOn is scheduled (or posted) to occur at the same time, the event with the smaller priority value will occur first. The two EventClass definitions are added directly below the Simulation element in Listing 40.

Listing 34: Definition of event classes

```
<EventClass Name="turnOn" Priority="5" />
<EventClass Name="turnOff" Priority="4" />
```

With event super-classes event classes can be organized into a generalization hierarchy. An event class may have multiple super-classes. This allows event sinks to be defined so that they trigger based on a more general event. In Listing 35 the turnOn and turnOff events have Switching as their sole super-class. The event sink named “Drain” will be triggered by either a turnOn or turnOff event (Listing 36), invoking the event method named Switching_Drain.

Listing 35: Definition of event classes using event super-classes

```
<Simulation>
  <EventClass Name="Switching" />
  <EventClass Name="turnOn" SuperClasses="Switching" Priority="5" />
  <EventClass Name="turnOff" SuperClasses="Switching" Priority="4" />
  <PowerMeter LocalID="PM1" DrainActivity="0">
    <EventSink Name="Drain" EventClasses="Switching" EventMethod="Switching_Drain" />
    <Script Type="text/javascript">
      function Switching_Drain(ev)
        {
        DrainActivity = 1;
        }
    </Script>
  </PowerMeter>
</Simulation>
```

Listing 36: Posting an item event that will trigger an event super-class

```
PostItemEvent(null, 'turnOn', 4, srmlCascadeNone)
```

9.10 Notifications

Notifications are pre-defined events that originate from a Simulation object. A Simulation object uses notifications to communicate with its host, giving the host an opportunity to do something in response to changes that take place inside the runtime environment. Some types of notifications tell the host when something will happen, or has happened to an item or item event. For example, the ItemCreated, and ItemDestroyed events occur at the beginning and ending of the lifetime of an item. Notifications about an item event would include EventPosted, and EventOccurred which describe when an asynchronous event was originated and when it was completed. Other notifications occur when the engine may need information from the host, such as LoadItemSource and LoadRandomNumberGenerator, which request that the host assist in the loading of an external item or random number generator. There are also notifications that simply provide information about the simulation itself, such as CurrentTimeSet, and StateChanged.
In Listing 8, the EventOccurred notification from the Simulation is being used to display details about the events as they occur in the simulation.

Another example of the use of notifications is in Listing 37. A custom random number generator named “OneTri” is defined in script. When the Simulation object needs to generate a new value from the “OneTri” distribution for the first time, the LoadRandomNumberGenerator notification is invoked and the script returns a new instance of the distribution.

Listing 37: RandomNumberGenerator Example

```
<html>
<head>
<title>Simulation Demonstration – RandomNumberGenerator</title>
<object id="Simulation" type="application/srml"></object>
<script type="text/javascript">
function RunSimulation()
  {
  Simulation.CurrentTime = new Date("1/1/2001").getVarDate ();
  Simulation.EndTime = new Date("1/2/2001").getVarDate ();
  var value = Simulation.Random("OneTri")
  alert (value)
  Simulation.Run();
  }
</script>
<script type="text/javascript">
function OneTri()
  {
  this.valueOf = function() { return 1.0 - Math.sqrt(Math.random()); }
  }
</script>
<script for="Simulation" event="LoadRandomNumberGenerator(Type, Arguments,
    RandomNumberGeneratorBox)" type="text/javascript">
  if (Type == "OneTri")
    RandomNumberGeneratorBox.Value = new OneTri();
</script>
</head>
<body>
<button onclick="RunSimulation()" type="text/javascript">Run Simulation</button>
</body>
</html>
```

10 ITEM CLASSES

While SRML allows properties and behavior to be defined for individual items, it also provides a generalization mechanism that allows such definitions to apply to multiple items. That mechanism is provided through the definition of item classes. An item class is analogous to class in object-oriented terms, in that it describes the properties and behavior for its instances while also providing a form of inheritance.

Any number of different item classes can coexist in a model by using an ItemClass element and specifying a unique Name for each one. Within the class definition is an instance prototype which is an embedded element with a tag name that matches the Name attribute of the item class. This prototype can have attributes with default values and data types validated by an XML schema.

Refer to Listing 38 for the general form of an ItemClass.

Listing 38: General form of an ItemClass definition

```
<ItemClass Name="itemtype1" SuperClasses="itemtype2 itemtype3 ... "
      Source="optional-source-uri" classproperty1="value1" ...>
  <Property Name="property2" Type="simpletype1" Default="value2"
       Placement="ClassOrElementOrAttribute"/>
  <Script Type="script-language" Source="optional-script-source"
      Placement="ClassOrInstanceOrIsolated">
    // Code ...
  </Script>
  <!-- The following element defines the prototype for the instances -->
  <itemtype1 property1="value3" ...>
    <Script Type="script-language" Source="optional-script-source">
      // Code ...
    </Script>
    <itemclass4 property2="value4" ...>
      ...
    </itemclass4>
  </itemtype1>
</ItemClass>
```

The highlighted area above indicates the prototype item, and the bold text shows the correspondence between the name of the prototype and the name of the item class.

The specific example in Listing 39 demonstrates the definition of an item class called Counter. Each of the ten instances created from this Counter class will have a method called Increment and will have their own Count value. As an item itself, the item class can also have properties and behavior that will be shared by the instances. The example defines an Instances property for the class that keeps track of the number of individual Counter instances. Any of the Counter instances can access its item class, through its intrinsic ItemClass property. Notice how the item accesses the Instances property in its item class in the code. In the note below, the code within the instance is directly updating the Instances value of the item class. Shared properties and behavior may be added to a specific item class in the same way add them to regular items, buy adding them outside the prototype. The Property elements placed before the prototype control the placement and data type of the item properties. When using these elements, the attributes placed on the item's elements become optional—they only provide an initial value.

Listing 39: Item class definition of a simple counter

```
<Simulation>
  <ItemClass Name='Counter' Instances='0'>
    <Property Name='Instances' Placement='Class' Type='int'/>
    <Property Name='Value' Placement='Instance' Type='int'/>
    <Counter Value='0'>
      <Script Type='text/javascript'>
      ItemClass.Instances++; // Note: Increment the item class's value.
      function Increment()
           {
           Value++;
           }
      </Script>
    </Counter>
  </ItemClass>
  <Counter Quantity='10' />
</Simulation>
```

10.1 Item Class Super-Classes

Sometimes a common set of properties and behaviors apply to more than one item class. In this situation a more general item class may be defined from which other more specific classes can inherit. SRML provides a capability found in traditional object-oriented languages for defining classes at various levels of generalization. The associated mechanisms include polymorphism, multiple-inheritance, and overriding in both item class instances and the item class itself. In the example below, a Motor has two more general super-classes: Asset and Operates. The SuperClasses attribute is used to specify the list of named item classes in descending priority.

In Listing 40, two item classes are defined: Asset, and Operates which have disjoint behavior. A third item class named Lamp inherits from Asset and Operates. Finally, ten lamps are defined which all have the behavior defined by Lamp item class.

Listing 40: Lamp item class defined using super-classes

```
<Simulation>
  <ItemClass Name='Asset' Count='0'>
    <Asset Operable='1'>
      <Script Type='text/javascript'>
        <![CDATA[
          ScheduleEvent(this, "failed", DateAdd("h", Random()*100, CurrentTime));
          function failed()
            {
            Operable=0;
            ScheduleEvent(this, "repaired", DateAdd("h", Random()*8, CurrentTime));
            }
          function repaired()
            {
            Operable=1;
            ScheduleEvent(this, "failed", DateAdd("h", Random()*100, CurrentTime));
            }
          ]]>
        </Script>
      </Asset>
    </ItemClass>
  <ItemClass Name='Operates'>
    <Operates Running='1'>
      <Script Type='text/javascript'>
        <![CDATA[
          function turnOn()
            {
            Running=1;
            }
          function turnOff()
            {
            Running=0;
            }
          ]]>
      </Script>
    </Operates>
  </ItemClass>
  <ItemClass Name='Lamp' SuperClasses='Asset Operates' />
  <Lamp Quantity='10' />
</Simulation>
```

11 MODEL INTEGRATION AND DISTRIBUTION

With the inherent modularity provided by XML, a single document may suffice for representing a large simulation. Many simulated items may take similar forms, and thus the mechanism of item classes provides the integration of common behavior. However, managing a single monolithic XML document does not work well when the pieces are developed independently, or when they need to be reused. External scripts provide a simple way to detach the behavior from the structure, thus allowing code reuse at the cost of some encapsulation. These scripts are specified by adding a Source attribute to a Script element. See Listing 41.

Listing 41: External script source

```
<Script Type='text/javascript' Source='Vehicle.js'/>
```

The entire behavior of an item or item class can thus be specified separately. The reason that encapsulation is slightly compromised is because the structural aspects of the item, being in XML, have been separated from the behavioral aspect, which is in a separate script. A more sophisticated technique known as embedded externals, allows portions of one simulation model to be extracted from another. This preserves encapsulation at the potential cost of a larger runtime footprint. Listing 42 shows several cases of embedded externals. In the first case, the vehicle is defined completely in the XML document: Vehicle.XML, which is similar to creating an instance from an external item class-many such identical vehicles can be added to the simulation model using the same source file. In the second case, the item class defined in the external file is included in the current model, thus making it available for instancing or sub-classing. The third case shows how separately compiled component-based objects, known as external items, are plugged into the simulation. External items coexist with items in a simulation model, and also provide the capability for entire simulation models to be connected together using interoperability layers such as CORBA, COM, and HLA.

Listing 42: Embedded externals

```
<Vehicle Source='http://srml.boeing.com/Vehicle.xml'/>
<ItemClass Name='Vehicle' Source='file:///Vehicle.xml'/>
<Vehicle Source='progid://simserver/VehicleComponent.Tank'/>
```