Proceedings of the 2004 Winter Simulation Conference
R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

SRML CASE STUDY: SIMPLE SELF-DESCRIBING PROCESS MODELING AND SIMULATION

Steven W. Reichenthal

Boeing
3370 Miraloma Ave.
Anaheim, CA 92803, U.S.A.

ABSTRACT

This paper provides an introduction to the Simulation Reference Markup Language (SRML) (SRML 2002) through a case study in which a simple self-describing process modeling and simulation representation is developed. In this context, "self-describing" refers to a simulation representation that includes not only a model's data, but also includes the behavioral semantics of the simulation objects, thereby enabling the execution of those models within a general-purpose simulation engine.

1 INTRODUCTION

SRML is an XML technology for infusing or otherwise describing the behavior of arbitrary XML data, using web-based techniques similar to those found in HTML. The language specification has been published as a note on the W3 Consortium web site with the goal of encouraging the development of common simulation interchange standards. Executing SRML requires an SRML simulation engine, which is software that combines a discrete-event simulation runtime environment with the XML Document Object Model (DOM), a scripting host, and a plug-in management system. A free run-time engine may be downloaded from Boeing.com (BOEING 2003) for evaluation and study.

Process modeling is an ideal medium for demonstrating SRML, because process flows can be represented declaratively with natural ease using XML. Likewise, the functionality underlying commercial process modeling and simulation software is often hidden; thus it is instructive to see how such functionality might be implemented. Many of the commercial tools employ a similar "factory" paradigm consisting of entities and blocks. For this case study, a simplified set of simulation objects are developed as a basis for study and extension.

2 BASIC CONCEPTS OF SRML

SRML should not be considered a programming language, but rather a composition language for integrating XML data models with behavior. The concept of SRML with its corresponding simulation engine is similar in concept to HTML with its corresponding web browser. Both environments are built upon the foundation of SGML and support scripting with plug-in extensibility. HTML's scripting capability has made it possible for web-pages to include "open source" functionality using arbitrary scripting languages like JavaScript/ECMAScript, or Python. Concordantly its "plug-in" capability provides for the execution of compiled, black-box functionality. Likewise, SRML was designed to include both of those features, but with the added capability for specifying classes of items and events using XML. Composition in SRML is provided by the ability for either a single xml file to contain all the data or for fragments of data to be assembled from files located at various locations.

3 PROBLEM STATEMENT

This case study is driven by a typical process-related problem that can be found in everyday America. Suppose our goal is to use simulation to model line queuing and the management of people at a public service facility such as a DMV. One responsibility for our simulation is that it must determine the average number of people that would be waiting in line at the information booth of our local DMV. We observe that people arrive at some rate per hour, and require on average several minutes of the clerk's time to determine where to send them next. A certain percentage of the time people are directed to one line or another. Though a problem this simple could be solved mathematically, the use of discrete-event simulation becomes more practical as complexity increases. The flow could be generalized and represented graphically as shown in Figure 1. Create1 represents the arrival of people, Process1 represents the activity involving a person and the information clerk, and Decide1 represents the one of two paths that the person will following the exchange.

A common approach to developing a simulation for a process flow problem is to employ the "factory" paradigm, which is based on the concept of the *entities* and *blocks*

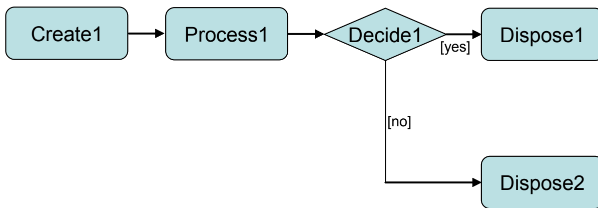

Figure 1: Sample Process

depicted in Figure 2. *Entities* are things that enter and exit *blocks* during the execution of a simulation. *Blocks* represent operations that process *entities*, and may be connected together to form a process flow. Four types of blocks are developed in this case study: *Create*, *Dispose*, *Process*, and *Decide*. The *Create* and *Dispose* blocks are the fundamental blocks that generate and terminate the existence of entities within a flow. A *Process* block receives an arriving entity and places it on a queue to wait for an available resource. Once a resource is available, the *Process* block simulates work using a random delay distribution for its duration. After the delay, the resource is released and the entity exits the *Process* block to be received by the next block or blocks in the flow. With the *Decide* block, an arriving entity takes either one path or the other in proportion to a specified probability. Each type of block has a set of properties that govern its operation, and the values of those properties may be varied in order to run experiments.

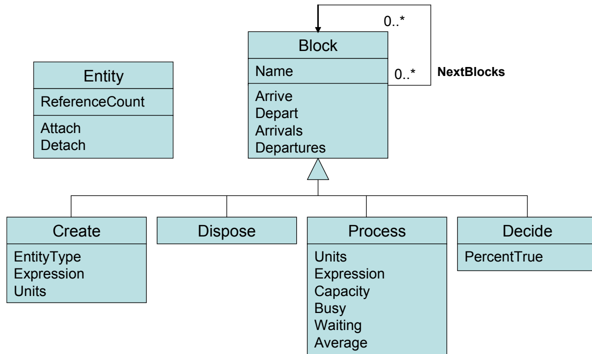

Figure 2: Factory Paradigm

4 XML REPRESENTATION OF THE SAMPLE PROCESS

Listing 1 is the XML that represents the process flow represented in Figure 1. Its schema was derived from a simple mapping of blocks to elements, properties to attributes, and relations to attributes. Creating the sample model first, without a schema affords the opportunity to experiment with different mappings. Other mappings are possible, each with benefits and consequences. However, the important point with respect to defining simulation behavior is that the schema may be arbitrary because SRML has constructs that allow the interchanging elements and attributes. An actual XML Schema (SCHEMA 2001) document could be developed to validate the model, however in this case one is not necessary.

Listing 1 - ProcessModel1.xml:

```
<ProcessModel ID="Factory1">
 <Create Name="Create1"
   NextBlocks="Process1"/>
 <Process Name="Process1"
   NextBlocks="Decide1"/>
 <Decide Name="Decide1"
   NextBlocks="Dispose1 Dispose2"/>
 <Dispose Name="Dispose1"/>
 <Dispose Name="Dispose2"/>
</ProcessModel>
```

This XML file could be loaded into an SRML simulation engine as is. The engine would merely create a hierarchically interconnected set of objects (items), because no behavioral descriptions have been provided. In this case, adding behavior means specifying what the *Create*, *Process*, *Decide*, *Dispose* tags do in the simulation—their operational semantics. Behavior may be added intrusively by modifying the file to either embed the behavioral markup within the file or to reference external behavior defined in another file. The advantage of the former is that the single file would be self-describing in terms of simulation; whereas the advantage of the latter permits different behaviors to be interchanged without having to modify the model's file.

Conversely, behavior may be added non-intrusively, yet still be self-describing by creating a separate simulation file that references both the model and the behavior, thereby permitting the model to remain independent from the simulation and behavioral definitions. This is the approach taken in the sample, and is shown in the Listing 2. To reference an external definition, SRML provides the Source attribute, which can be added to any element in an XML document. When the simulator encounters a Source attribute, it attempts to load and map the external definition to the item described by the element. The simulator recognizes the Source attribute and loads ProcessLibrary.xml, which defines the behavior.

Listing 2 - ProcessSimulation1.xml:

```
<srml:Simulation
  xmlns:srml="urn:x-schema:srml.xdr"
  xmlns="">
 <ProcessLibrary
   srml:Source="ProcessLibrary.xml"/>
 <ProcessModel
   srml:Source="ProcessModel1.xml"/>
</srml:Simulation>
```

5 PROCESS LIBRARY

Rather than embedding all the behavior into a single XML file, the process library file shown in Listing 3 is a composition of behavior from several separate files. Separating the files makes it convenient to modularize the components of the library for development and extension.

Listing 3 - ProcessLibrary.xml:

```
<ProcessLibrary xmlns=""
   xmlns:srml="urn:x-schema:SRML.xdr">
 <srml:ItemClass Name="Entity"
   Source="Entity.xml"/>
 <srml:ItemClass Name="Block"
   Source="Block.xml"/>
 <srml:ItemClass Name="Create"
   Source="CreateDispose.xml"/>
 <srml:ItemClass Name="Dispose"
   Source="CreateDispose.xml"/>
 <srml:ItemClass Name="Process"
   Source="Process.xml"/>
 <srml:ItemClass Name="Decide"
   Source="Decide.xml"/>
</ProcessLibrary>
```

6 ENTITY CLASS

Entities, which are implemented as objects that are created and destroyed during the execution of the simulation, are defined using the SRML *ItemClass* construct, see Listing 4. Generally speaking, an item class defines a class of items with common properties, structure, and behavior—much like the ordinary class construct provided by an object-oriented programming language. Each property has a name, an optional type, and an optional default value. The type and default value are optional because an XML Schema may already have provided those definitions. A single property is defined on the Entity item class, reference count, which keeps track of the number of blocks that are operating on the entity. Specific behaviors developed for entities include the *Attach* and *Detach* method. The *Attach* method simply increments a reference count on the entity, whereas the *Detach* method decrements the reference count and destroys the item when the count reaches zero. Notice that the default value for the reference count is 1, which means that an initial attach is not necessary. Therefore, for this to work properly every next block must attach to the entity before the previous block detaches.

Listing 4 - Entity.xml:

```
<srml:ItemClass Name="Entity"
  xmlns:srml="urn:x-schema:SRML.xdr">
 <srml:Property Name="ReferenceCount"
   Type="i4" Default="1"/>
 <srml:Script Type="text/javascript"
   Placement="Isolated">
  <![CDATA[
  function Attach ()
   {
   this.ReferenceCount++
   }
  function Detach ()
   {
   if (this.ReferenceCount &&
        --this.ReferenceCount == 0)
    this.DeleteItem (this)
   }
  ]]>
 </srml:Script>
</srml:ItemClass>
```

Item classes are not intended to replace the use of traditional class definitions. Actually, a simpler alternative design for the process library would not even need to create an entity class or have any instances by simply managing entity counts, however, such as design would not scale well. Using an item class allows the simulation engine to manage the object's lifetime and persistence. Script placement for the *Attach* and *Detach* methods is specified as Isolated, which means a single script is created to service all instances, but that script is isolated from the runtime environment and must use "this" to access the particular instance.

7 BLOCK CLASS

The Block item class shown in Listing 5 serves as the general base class for specific types of blocks. Using this class, all blocks inherit a *Name* property which keeps track of the number of entity arrivals and departures, and manages the links among blocks. The *Name* property uniquely identifies each block within a local scope, and is defined with a data type of *srml.LocalID*, thereby allowing each block instance to have a unique name according to some local scope. In this case, the local scope is the entire model, but in a larger model the same block name could potentially be used more than once within different scopes. The *NextBlocks* property is of type *srml.Links*, which instructs the simulator to manage the contents as a list of related objects. Default methods exist to handle the arrival, departure, attachment and detachment of entities in a standard way. These methods may be overridden in the sub-classes. The *Arrive* method is the most likely method that a block will override, and, by default, this method simply causes the entity to immediately depart. The *Depart* method by default causes the entity to arrive at all of the next blocks. To attach an entity, the simulator's *SendEvent* method is called, to make a synchronous invocation of the entity's *Attach* method. *PostEvent*, would have worked just as well, although it makes an asynchronous invocation through the simulator's event list. The *Placement* attribute on the script is set to "Instance", specifying that each instance will have its own script.

Listing 5 - Block.xml:

```
<srml:ItemClass Name="Block"
  xmlns:srml="urn:x-schema:SRML.xdr">
 <srml:Property Name="Name"
   Type="srml.LocalID"/>
 <srml:Property Name="NextBlocks"
   Type="srml.Links"/>
 <srml:Property Name="Arrivals" Type="i4"/>
 <srml:Property Name="Departures" Type="i4"/>
 <srml:Script Type="text/javascript"
   Placement="Instance">
  <![CDATA[
  function Entity_Attach (objEntity)
   {
   Arrivals++
   SendEvent (objEntity, "Attach")
   }
```

```
function Entity_Detach (objEntity)
 {
 PostEvent (objEntity, "Detach")
 Departures++
 }

function Arrive (objEntity)
 {
 Entity_Attach (objEntity)
 Depart (objEntity)
 }

function Depart (objEntity)
 {
 for (var i = 0, n = NextBlocks.Count;
       i < n; i++)
  PostEvent (NextBlocks (i), "Arrive",
    objEntity)
 Entity_Detach (objEntity)
 }
]]>
 </srml:Script>
</srml:ItemClass>
```

8 CREATE AND DISPOSE BLOCKS

Both the *Create* and *Dispose* blocks are defined in the same file as belonging to a collection of item classes specified with an *ItemClasses* element, as shown in Listing 6. Entities are manufactured with a *Create* block according to a recurring pattern and are sent through the linked next blocks. Behavior corresponding to the *Create* tag is defined in an item class named *Create*, and this class has the Block class as its only super-class—SRML permits an item class to have multiple super-classes. The *EntityType* property controls which type of entity to create and by default has the value “Entity”. Having the *EntityType* property makes it convenient to create and use sub-classes of the Entity class with unique properties or behavior. The Generate method is used for scheduling the generation of a new entity. In turn, it uses the simulator’s *ScheduleEvent* method to schedule the invocation of the *Generated* method at a random time. A random distribution may be specified using the *Expression* property, which holds a string that names a random distribution and its corresponding parameters. This property defaults to a random exponential distribution with a parameter of 1. The string is supplied to the simulator’s *Random* function when scheduling the next item to be generated. The first call to *Generate* is placed outside of any function is called when the script for the process block is first created—like code in a constructor. Within the *Generated* method, a call to *CreateItem* is made which will create a new entity according to the specified *EntityType*. Each new entity is placed at the root of all items indicated by *Simulation.Object* in the second parameter to *CreateItem*.

The simplest of the blocks the *Dispose* block. Like the *Create* block, its super-class is the *Block* class, however, it overrides the *Arrive* method with an empty procedure definition. Thus, when an entity arrives at a *Dispose* block, its reference count will not be incremented and nothing will happen, which allows the entity to delete itself when the previous block detaches and no other blocks are holding references.

Listing 6 - CreateDispose.xml:

```
<srml:ItemClasses
  xmlns:srml="urn:x-schema:SRML.xdr">
 <srml:ItemClass Name="Block"
   Source="Block.xml"/>
  <srml:ItemClass Name="Create"
    SuperClasses="Block">
  <srml:Property Name="EntityType"
    Type="string" Default="Entity"/>
  <srml:Property Name="Expression"
    Default="Exponential 1"/>
  <srml:Property Name="Units" Default="h"/>
  <srml:Script Type="text/javascript"
    Placement="Instance">
   <![CDATA[
   Generate ()

   function Generate ()
    {
    var t = DateAdd (Units,
     Max (0, Random (Expression)),
     CurrentTime)
    ScheduleEvent (this, "Generated", t)
    }

   function Generated ()
    {
    var objEntity = CreateItem (EntityType,
      Simulation.Object)
    Depart (objEntity)
    Generate ()
    }
   ]]>
  </srml:Script>
 </srml:ItemClass>

 <srml:ItemClass Name="Dispose"
   SuperClasses="Block">
  <srml:Script Type="text/javascript"
    Placement="Instance">
   <![CDATA[

   function Arrive (objEntity)
    {
    }

   ]]>
  </srml:Script>
 </srml:ItemClass>
</srml:ItemClasses>
```

9 PROCESS BLOCK

The *Process* block is used to model an activity performed on an entity in which resources are required to be allocated before the activity can begin. Once begun, the activity consumes some amount of time until completion, at which point resources are released and the entity departs. Process blocks, as defined in Listing 7, have a limited quantity of resources (Capacity) which can service (as “Busy”) only a

single entity at a time. Another simplification is that capacity remains constant over time. When an entity arrives, a resource may be allocated immediately if available. To simulate the performance of the activity, the process causes a delay to occur. The delay is calculated using a random value from a specified distribution, as specified in the *Expression* property. After the delay period, the resource is released from its busy state, and the entity departs to the linked blocks to which the process connects. It may be possible that no resources are available when an entity arrives at a process block. In this situation, the entity is added to a first-in-first-out *Waiting* queue to be serviced when a resource becomes available, during the *Release* operation. This example also shows how it is possible for the script to be in a separate file (see Listing 8) by using the *Source* attribute.

Listing 7 - Process.xml

```
<srml:ItemClasses
  xmlns:srml="urn:x-schema:SRML.xdr">
 <srml:ItemClass Name="Block"
   Source="Block.xml"/>
 <srml:ItemClass Name="Process"
   SuperClasses="Block">
  <srml:Property Name="Units" Default="h"/>
  <srml:Property Name="Expression"
    Default="Triangular 0.5 1 1.5"/>
  <srml:Property Name="Capacity" Type="i4"
    Default="1"/>
  <srml:Property Name="Busy" Type="i4"
    Default="0"/>
  <srml:Property Name="Waiting"
    Type="SR_Collections.SRList"/>
  <srml:Property Name="Average"
    Type="SR_MLTools.TimeAverage"/>
  <srml:Script Type="text/javascript"
    Placement="Instance"
    Source="Process.js"/>
 </srml:ItemClass>
</srml:ItemClasses>
```

Listing 8 - Process.js

```
function Arrive (objEntity)
 {
 Entity_Attach (objEntity)
 Seize (objEntity)
 }

function Seize (objEntity)
 {
 if (Busy < Capacity)
  {
  Busy++
  Delay (objEntity)
  }
 else
  Waiting.Add (objEntity)
 Average.AddValue (Waiting.Count,
   CurrentTime)
 }

function Delay (objEntity)
 {
 var t = DateAdd (Units,
   Max (0, Random (Expression)), CurrentTime)
 ScheduleEvent (this, "Release", t,
   objEntity)
 }

function Release (objEntity)
 {
 if (Waiting.Count > 0 && Busy <= Capacity)
  {
  var objEntityT = Waiting.Remove (0)
  Delay (objEntityT)
  }
 else
  Busy--
 Average.AddValue (Waiting.Count,
   Simulation.CurrentTime)
 Depart (objEntity)
 }
```

A *Decide* block receives arriving entities and sends them to one of two next blocks according to some proportion specified in the using *PercentTrue* property. The *Arrive* method evaluates a uniform random value and compares it with the value in the *PercentTrue* property. If the random value is less than the percent true, the entity departs to the next block at index zero, otherwise it departs to the next block at index 1. The default value for *PercentTrue* is .5, so that half of the entities will take either path. The code is shown in Listing 9.

Listing 9 - Decide.xml

```
<srml:ItemClasses
  xmlns:srml="urn:x-schema:SRML.xdr">
 <srml:ItemClass Name="Block"
   Source="Block.xml"/>
 <srml:ItemClass Name="Decide"
   SuperClasses="Block">
  <srml:Property Name="PercentTrue"
    Type="r4" Default="0.5"/>
  <srml:Script Type="text/javascript"
    Placement="Instance">
   <![CDATA[

   function Arrive (objEntity)
    {
    Entity_Attach (objEntity)
    var t = Random("Uniform")
    if (t <= PercentTrue &&
      NextBlocks.Count > 0)
     PostEvent (NextBlocks (0), "Arrive",
       objEntity)
    else if (NextBlocks.Count > 1)
     PostEvent (NextBlocks (1), "Arrive",
       objEntity)
    Entity_Detach (objEntity)
    }

   ]]>
  </srml:Script>
 </srml:ItemClass>
</srml:ItemClasses>
```

10 CONCLUSION

XML has become a popular format for representing data in an open fashion, but the processing of that data is often compiled into custom application programs thereby cou-

pling the visible data with hidden operations. With the use of XML Schemas, data becomes self-describing with respect to structure, and with SRML data also becomes self-describing with respect to behavior. A simple set of process modeling objects was developed in this case study in order to demonstrate the basic concepts of SRML with its ability for adding simulation behavior to XML data. Additionally, those process objects may serve as a basis of extension and improvement.

Recently, the Simulation Interoperability Standards Organization (SISO) has established a Product Develop Group (PDG) with the objective to develop a specification for Base Object Models (BOMs) (BOM 2003) to be used for defining patterns of interplay among simulation components. In the future, an industry standard catalogue of behavioral definitions for interchangeable process simulations could exist, and the development of BOMs may be a first step in the process towards that end. Specifically, SRML can serve to produce self-describing models supporting the behavior associated to the conceptual entities and processes of a BOM in a platform neutral manner. In this light, SRML would help to facilitate the simulation composability that BOMs offer.

REFERENCES

BOEING. 2003. SRML - Simulation Reference Simulator Evaluation, available online via <`http://www.boeing.com/assocproducts/srml/`> 2003

BOM. 2003. Base Object Model (BOM) Template Specification Volume I - Interface BOM, SISO-STD-003.1-DRAFT-V0.9

SCHEMA. 2001. Schemas express shared vocabularies and allow machines to carry out rules made by people <`http://www.w3.org/XML/Schema/`> [Accessed May 2, 2001]

SRML. 2002. SRML - Simulation Reference Markup Language W3C Note, <`http://www.w3.org/TR/SRML/`> [Accessed December 18, 2002]

AUTHOR BIOGRAPHY

STEVE REICHENTHAL Steven W. Reichenthal is an Associate Technical Fellow at the Boeing Company. He develops simulations and software applications as a member of the Logistics Engineering organization in Anaheim, California. He has a Masters degree in Computer Science and an MBA, and teaches software development courses at the California State University in Fullerton as an adjunct professor.