Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

SIMULATIONS ON .NET USING HIGHPOINT'S HIGHMAST™ SIMULATION TOOLKIT

Peter C. Bosch

Highpoint Software Systems, LLC
S42 W27451 Oak Grove Lane
Waukesha, WI 53189, U.S.A.

ABSTRACT

This paper describes the philosophy, architectures and key features of a new .Net-based simulation object model and toolkit called HighMAST™ (Highpoint Modeling and Simulation Toolkit). HighMAST™ is a set of class libraries built on top of Microsoft's .Net platform. It was built to take advantage of the object-oriented flavor and extensive integration plumbing ingrained in the .Net framework. It supports "active entity", "block-based", "workflow-oriented" and several other types of simulation architectures in both the discrete-time and continuous domains. And it enables developers to approach their simulation frameworks or applications in a wide range of languages including such inexpensive and available languages as C# and VB.Net.

1 INTRODUCTION

A number of criteria apply when selecting a simulation platform. Depending on the type of project, the type of organization executing the project, and the type of staffing model being applied to the project, each criterion may have different weighting in the final selection process.

The platform must provide a good foundation for the simulation to be developed. This means that the basic engine must be sufficiently fast, comprehensible and accessible. As well, the engine should be supplemented with a well-designed and feature-rich set of libraries that provide substantially all of the primitive constructs that the simulation designers need. Finally, the libraries should be readily extensible, to enable those constructs that are more unique to the specific domain to be modeled.

The platform should support rapid, modern application development. Larger-scale simulations require esoteric core simulation programming, but also entails programming in less specialized domains such as database access, animation and application-integration code. In order to inexpensively support this range of needs, its language should be general-purpose, development tools should be broadly available and robust across all of the uses to which it will be placed, and the non-simulation skills needed to construct the desired applications should be reasonably available.

The platform should support a range of simulation architectures and approaches. Simulations' characteristics vary widely from one application to the next, and often for multiple applications within the same enterprise suite. A simulation framework that supports only the "live-entity" mechanism in which, for example, a customer is modeled with a thread, will break down when faced with a system of thousands of customers. A mechanism in which entity flow is modeled by objects passing through "blocks" will break down when faced with a large number of entities, and is sometimes better modeled using a continuous flow metaphor. The platform should support all of these approaches.

This paper describes a simulation platform called HighMAST™, which was undertaken with a design philosophy intended to address the three preceding needs – foundation solidity, modern development methods, and architectural freedom. It's core engine is written in C# using high performance algorithms and focused multithreading capability for performance, and an object oriented architecture for comprehensibility and accessibility. Full-featured libraries include queues and events, resources and resource pools, collection classes of many types, PERT and CPM analysis modules, executives, and a base-level model class with a highly customizable application state machine. Simulations already built on HighMAST™ have extended existing classes into specific simulation domains such as batch manufacturing.

The remainder of this paper describes specifics of the points made above. Sections 2 and 3 describe the foundation provided by HighMAST™, sections 4 through 7 discuss the strengths with which HighMAST™ supports large scale application development, and sections 8 through 13 describe the technologies and architectural choices enabled through the HighMAST™ platform. Finally, section 14 provides the reader with a summary of the strengths of this powerful newcomer to the field of modeling and simulation, and suggests some sources of further information.

2 BASE ENGINE

The base engine behind HighMAST™ consists of two orthogonal pieces, the executive and the model. The executive is a C# class with several supporting classes, that performs event (callback) registration and sequencing. In essence, through the executive, a model entity may request to receive a callback at a specific time, with a specific priority, on a specified method, and with a specified object provided to that method on the callback. The entity may rescind that request at any point before the call, and the method need not be located on the entity requesting the callback. Further, the entity requesting the callback may select how the callback is to be handled, currently among three choices:

1. **Synchronous** – the callback is called on the dispatch thread, and upon completion, the next callback is selected based upon scheduled time and priority. This is similar to the "event queue" implementations in Garrido (1993) and Law and Kelton (2000).
2. **Detachable** – the callback is called on a thread from the .Net thread pool, and the dispatch thread then suspends awaiting the completion or suspension of that thread. If the event thread is suspended, an event controller is made available to other entities which can be used to resume or abort that thread. This is useful for modeling "intelligent entities" and situations where the developer wants to easily represent a delay or interruption of a process.
3. **Batched** – all events at the current time and priority are called, each on separate threads, and the executive, except for servicing any new events registered for that time and priority, awaits completion of all running events. This may bring about higher performance in cases such as battlefield and transportation simulations where multiple entities may sense current conditions, plan and execute against that plan.

The code below describes the API through which event requests are registered.

```
// public member of Executive class.
public long RequestEvent(
  ExecEventReceiver eer, // user callback
  DateTime when,
  double priority,
  object userData,
  ExecEventType execEventType){ … }
```

The Model class provided with HighMAST™ performs containment and coordination between the executive, the model state machine and model entities such as queues, customers, manufacturing stages, transport hubs, etc.

The model's state machine is used to control and indicate the state of the model – for example, a model that has states such as design, initialization, warmup, run, cooldown, data analysis, and perhaps pause, would represent each of those states in the state machine. Additionally, the application designer may attach a handler to any specified transition into or out of any given state, or between two specific states. Handlers may be given a sequence number to describe the order in which they are to be executed. Each transition is performed through a two-phase-commit protocol, with a prepare phase permitting registrants to indicate approval or denial of the transition, and a commit or rollback phase completing or canceling the attempted transition. The following code describes the interface that is implemented by a transition handler. User code may implement any of the three delegates (API signatures) at the top of the listing, and add the callback to the handlers for transition out of, into, or between specified stages.

```
public delegate ITransitionFailureReason
  PrepareTransitionEvent(Model model);
public delegate void
  CommitTransitionEvent(Model model);
public delegate void RollbackTransitionEvent(
  Model model, IList reasons);

public interface ITransitionHandler {
    event PrepareTransitionEvent Prepare;
    event CommitTransitionEvent Commit;
    event RollbackTransitionEvent Rollback;
    bool IsValidTransition { get; }

    void AddPrepareEvent(
      PrepareTransitionEvent pte,
      double sequence);
    void RemovePrepareEvent(
      PrepareTransitionEvent pte);
    void AddCommitEvent(
      CommitTransitionEvent cte,
      double sequence);
    void RemoveCommitEvent(
      CommitTransitionEvent cte);
    void AddRollbackEvent(
      RollbackTransitionEvent rte,
      double sequence);
    void RemoveRollbackEvent(
      RollbackTransitionEvent rte);
}
```

3 LIBRARIES

The HighMAST™ core includes foundation libraries that provide a conceptual base for those familiar with software model building. In addition to the executive and model classes, these libraries include distributions of various types, a pulse source, an item generator, many simulation-item-specific queue, list, and collection-related classes, a resource management mini-framework and a wide range of other simulation primitives.

These primitives are designed with common patterns and good object-oriented architecture and development in mind. They make wide but appropriate use of inheritance for ease-of-implementation, interfaces for architectural orthogonality, and events for easy integration of disparate types of objects.

There are a number of higher-level capabilities, some described in sections 9, 10 and 0, that are built on these primitives – this construction has served as a proving ground for the design and development work done on the basic constructs. Furthermore, we have now constructed an entire batch manufacturing simulation framework on top of even these higher-level constructs, further demonstrating the extensibility and adaptability of this architecture.

4 DEVELOPMENT TOOLS

HighMAST™ is built on Microsoft's .Net libraries and Common Language Runtime (CLR). Core code is written in C# (C-sharp) and extension libraries and tools can be written in any of a wide variety of applicable languages. Of course, development can take place within Microsoft's excellent Visual Studio development environment, and leverage all of the tools that environment is made to integrate with, from bug-trackers and source code control systems to GUI, database and other components from third-party developers. Most importantly, the effort to leverage this tool base is minimal, since you're working in a well-known, well-used and well-supported environment.

With a simulation (or any application) built on a broad, well-supported technology base, an architect can typically count on the availability of a wide range of graphical, database, reporting and other pre-built tools and components for integration into his application.

Finally, since we expect to have a graphical development environment soon, we have used attributes (essentially, descriptors attached to code constructs such as fields and methods) to identify the purposes of those code constructs so that, for example, ports and connectors can indicate their directionality, objects can declare a preference of custom explorer type, and a model's state machine can be self-documenting to a GUI.

5 DEVELOPMENT METHODOLOGIES

Whether a web application, desktop application, database application or other type, a simulation is first and foremost, an application. In many organizations nowadays, legacy applications are the only ones that are permitted to live as standalone desktop applications. Web applications are being developed with an eye towards integration through portals, wrappers or web services, and new enterprise applications today are becoming larger and more strategic as supporting technologies mature. All of this makes it highly likely that a simulation written today will have integration requirements and functional scope far beyond those of a simulation application written only a few short years ago.

On the thesis that a simulation is an application, and that integration and larger-scale development were critical factors, we chose .Net as the foundation technology. Firms already know, or can learn readily, how to use .Net for broad-based application development. An architect defines subsystems and interfaces. Services (including simulation) are designed and written by one or more groups of developers specializing in the particular area of development. Databases are designed and written by database experts. User interfaces are designed by usability experts and developed by GUI developers. Integration and testing follow well-understood paths, and the application is rolled out in much the same way that the rest of an organization's applications are rolled out – or perhaps better.

This seemed to us to be far superior to the "Guess-and-Go-Dark" approach taken by many simulation experts with specialized and esoteric languages, limited and simulation-centric tools and wide experience in delivering applications that require the aid of a highly-trained operator in order to be useful.

6 RELATED TECHNOLOGIES

A number of technologies are deeply embedded in Microsoft's .Net technology base, and are therefore available to - and in most cases, used by, HighMAST™. Furthermore, each of these technologies, through the base-level class libraries is a part of the fabric of HighMAST™, free of the often-ignored friction of "integration".

XML is a broad-based industry standard that is central to many aspects of Microsoft's .Net initiative. Its principal purpose is to provide self-documenting data structures that can be manipulated by entities without full knowledge of the entire form of the data. It is useful for integration and serialization, as well as being a key technology to other technologies such as SOAP and ADO.Net. Through .Net's XML implementations, HighMAST™ can provide excellent and broad support for browsing, reading and writing XML documents.

Attributes are a mechanism for describing some aspect of a piece of code. They can be attached to methods, properties, interfaces, classes, namespaces and other artifacts. The descriptions can be used to specify indigenous aspects of a piece of code such as it's a class' serializability or a static variable's being thread-local, or can be used to declare some custom aspect of the code such as whether a piece of code represents a model or view construct, and perhaps whether a particular property represents an input port, output port or bidirectional port in some piece of code. Through the use of attributes, HighMAST™ is prepared to indicate to tools, graphical and otherwise, the intent of critical code elements such as tasks, ports, connectors and resources.

Delegates are an object-oriented incarnation of the function pointer. They represent callbacks, but the multicast delegate goes a step further and permits a single delegate instance to perform callbacks on more than one target object. HighMAST™ uses delegates as events (see below) as well as using them as targets for scheduling callbacks.

See below for the signature that a method must follow in order to be able to act as a schedulable callback.

```
public delegate void ExecEventReceiver(
  IExecutive exec,
  object userData);
```

This means that a method can have any name, and must receive two arguments, one an instance of IExecutive and the other, a generic object that may be used to contain any desired user data or may be null. Any method on any object that follows this criterion, may be used as a target for a scheduled event.

Events are a construct that uses a multicast delegate to collect callbacks that are interested in being called when a particular occurrence happens. Their syntax is shown in the listing below.

```
// Add a handler to the status change event
// fired by the object '_order'. The method
// void StatusChangeHandler(…) is declared
later.
_order.StatusChangeEvent+=
    new PropertyChan-
geEvent(StatusChangeHandler);
```

Events provide a simple and powerful mechanism for tying disparate elements of a simulation together while minimizing dependencies between them.

SOAP and Web Services are technologies that enable an application, running on a web server, to behave like a library for other programs to call into by way of a network. This is a big part of Microsoft's "vision" for .Net, and provides a very powerful integration point for, among other things, enterprise applications that desire to use simulation-oriented services. Highpoint has several demonstrations of HighMAST™ simulations exposed to the web via SOAP and web services.

.Net Framework Libraries – From the System.Collections namespace with its ArrayList, Hashtable, Queue, Stack and and SortedList, to the System.Threading namespace with its behind-the-scenes threadpool, the .Net Framework Libraries provide some impressive raw material for creating customer-pleasing capabilities.

Built-in Compiler and Code DOM – The framework libraries provide, in C#, an object model called the "Code DOM (Document Object Model)" for constructing source code and subsequently compiling it. Integrated into a simulation environment, this provides a simple way for simulation system designers to support user-provided behavior. HighMAST™ includes a component called an EvaluatorFactory, that takes a string that represents a code snippet from the user, compiles it at run-time to generate an Evaluator and integrates in into a running application.

Application Domains – Server development on Windows used to involve a choice between the stability risks of sharing memory space between different clients' applications, and the performance risk of creating a new process for handling each client's request. Application domains are a simple way of separating clients' computation so that a crash in one does not take down all of the others, and yet static data in the application are kept isolated between clients.

Application domains are probably best used to isolate different simulations running on the same server, but whose results are intended for different users.

The Common Language Runtime (CLR) is a virtual machine environment in which .Net code is run. All code is stored in the same intermediate form, MSIL (Microsoft Intermediate Language) but runs as compiled code, with the compilation happening at load time. This means that while the engine and core libraries are written in C# for performance and advanced capability, application specific code and components can be written in VB.Net or another .Net language that your IT or engineering staff prefer.

Unmanaged Code – The CLR provides many mechanisms to protect programmers from making mistakes like walking off the end of an array, dereferencing a wild pointer, forgetting to free memory after it is no longer needed, and many others. This is a productivity boon, but comes at a performance cost. By declaring a particularly performance sensitive region of code to be "unmanaged", the developer can choose to skip these protective measures and achieve performance closer to that of C and C++.

ADO.Net is Microsoft's premier data access technology. It can be used to access data from a database, of course, but it is also capable of reading from, and writing to, spreadsheets, textual flat files, and XML documents using the same coding structures and syntax. ADO.Net DataSets contain tables, rows, views, relationships and many other object-oriented constructs that permit the application to create and use a local in-memory database.

NUnit Integration – While not a Microsoft technology, NUnit is an excellent testing framework distributed from NUnit's website at http://www.nunit.org/. HighMAST™ is written with many tests that use this framework, and we advocate the use of NUnit in any larger-scale .Net software project.

7 *NIX CAPABILITIES THROUGH MONO

An organization called Ximian.org has had between 30 and 150 open source developers working on a project called Mono since about July of 2001. Currently, about 150 have CVS commit access. Mono is an open source implementation of .Net capable of running on Linux, FreeBSD and Windows (with the XP/NT core). There is also an interpreter, which is slower, that runs on the s390, SPARC and PowerPC architectures.

The Mono distribution includes a compiler for the C# language, a runtime for the Common Language Infrastructure (analogous to Microsoft's Common Language Runtime) and a set of class libraries. The runtime can be embedded into your application. It has an implementation of ASP.NET and a limited ADO.NET implementation as part of its distri-

bution. While there are some questions about Microsoft's strategy regarding Mono, the class libraries are X11 licensed (similar to the BSD license), so anyone can use and change them without worry. The runtime libraries are LGPL, so you can ship proprietary applications linked against Mono. Finally, the compilers and utilities (varies by utility) are typically licensed under the Gnu Public License.

Many software projects have reported minimal issues porting from .Net to Mono, and Highpoint is considering plans to expand our platform support to include full support for Mono under our core libraries.

8 MODEL PARTICIPANT CHOICES

Most modeling environments have first class objects. In many, they are called blocks and are "wired together", in some they are called "active entities" or "processes", Law and Kelton (2000), Watkins (1993), and embody a thread of execution, and in some they are domain-level macro objects whose relationships are specified in the creation of the simulation design.

With HighMAST™, you have a choice of any or all of the above. Active entities can be created using the detachable callback construct. Block-and-port models can be created using the port-equipped primitives we already have, or by building your own through a simple recipe. Domain specific models can be created using macro objects such as our SupplyChainMail™ objects.

9 WORKFLOW ARCHITECTURE

HighMAST™ includes a sophisticated workflow engine that permits a designer to describe networks of tasks to any level of detail, and then invoke them on an arbitrary number of simultaneous or time-staggered subjects, modeling the execution of a number of similar tasks all of which contend for a (limited) number of resources, for example.

This workflow engine can be used to model complex hierarchical procedures. Procedures can be specified at a high level, and later provided with detailed sub-tasks to increase fidelity where more detailed exploration is desired. **Figure 1** depicts the concept of a nestable task graph.

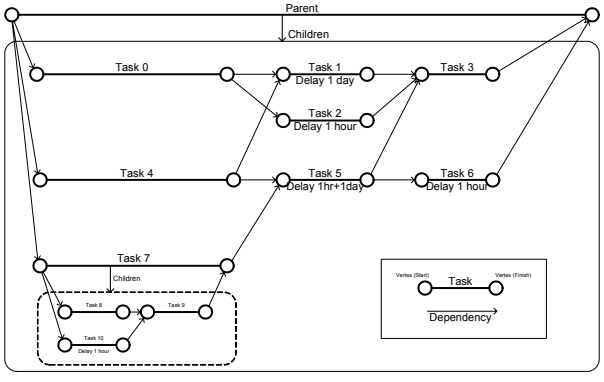

Figure 1: Nestable Task Graph

As the 'parent' task is begun, each of tasks 0, 4 and 7 are notified that they may begin. When task 0 completes, it notifies tasks 1 & 2 of its completion. Task 2 has no other predecessors, so it may run, but task 1 requires completion of task 4, too, so it does not run until it is informed of task 4's completion.

This construct, and the means for critical and shortest path determination are shown in Cormen et al. (2001).

A designer may use inheritance from the Task class, or utilize a set of events and an external completion signaler to fully integrate his domain-specific tasks into the workflow metaphor. Each task may use the scheduler, have self-determined duration, acquire & release resources, interact with other tasks, etc.

Several types of relationships are provided to ensure sequence and simultaneity as required by the designer. These relationships are depicted in **Figure 2** below.

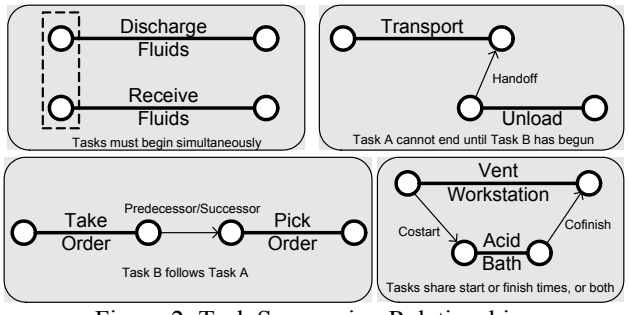

Figure 2: Task Sequencing Relationships

From the top left, going clockwise, the first (synchronizer) relationship ensures that two tasks receive start signals at the same clock time. The second (handoff) relationship ensures that the 'unload' operation must have begun before the 'transport' operation can conclude. The third (predecessor/successor) relationship ensures that the 'pick order' task does not begin until the 'take order' task has concluded. The fourth box shows two relationships, the first a costart relationship, which ensures that the 'acid bath' cannot start until the 'vent workstation' task has begun. The second relationship is a cofinish relationship, which ensures that the 'vent workstation' task cannot finish until the 'acid bath' task finishes.

10 RESOURCE MANAGEMENT

HighMAST™ resources implement a simple interface, shown below. They have a Capacity field, representing how much they can give, and an Available field, representing how much remains to be given. They also support several means of processing resource requests, and fire several events, meaningful to the lifecycle of a resource.

```
public interface IResource : IModelObject {
    IResourceManager Manager { get; set; }
    double Capacity { get; }
    double Available { get; }

    bool Reserve ( IResourceRequest request );
    void Unreserve ( IResourceRequest request );
    bool Acquire ( IResourceRequest request );
    void Release ( IResourceRequest request );

    event ResourceStatusEvent AcquiredEvent;
    event ResourceStatusEvent ReleasedEvent;
}
```

HighMAST™ includes a generic Resource implementation that can be configured as integral (capacity and acquisitions are both integral – models discrete resources), atomic (integral, but capacity is 1.0 – models singleton resources), or neither (capacity and acquisitions are unconstrained – models partial acquisition of scalar resources, such as pounds of steam drawn off of a steam system.)

Resource requests serve as a means for processing a request, as well as a means for tracking the resource pool from which an acquisition was made, and finally for executing a release. The requester communicates his desired quantity, and is informed of the granted quantity through the resource request. The GetScore(…) method holds the requester's logic for determining the score of each available resource, and the Resource Manager grants the best resource. A resource that scores double.MinValue will never be granted to a requester, and a resource that scores double.MaxValue will immediately be granted.

```
public interface IResourceRequest {

  double GetScore( IResource resource );
    double QuantityDesired { get; }
    double QuantityObtained { get; set; }
    IResource ResourceObtained { get; set; }
    IResourceManager
      ResourceObtainedFrom { get; set; }

    bool Reserve(
      IResourceManager resourceManager,
      bool blockAwaitingAcquisition);
    bool Acquire(
      IResourceManager resourceManager,
      bool blockAwaitingAcquisition);
    void Release();
}
```

11 PORTS-AND-CONNECTORS

HighMAST™ includes a Port-and-Connector architecture that allows the developer to create a familiar "create objects with ports and wire the ports together with connectors" model. There are very few constraints placed on the kind of object that can serve as owner of such ports – basically, it must be able to enumerate ports, and inform the ports of their owner through a registration protocol.

Ports serve to transfer data objects between entities such as products between assembly stations in a manufacturing simulation. They present objects to, and accept objects from, the world outside of a port owner. They fire events when data arrives or is sent, and support out-of-band data, which is a set of alternate date elements that usually pertain to the data object being transmitted. As an example, a vat, transferring 100 kg of sucrose mixture to another vat, might present a Mixture object on its output port, and a TimeSpan object on an out of band channel called "MinTransferDuration".

Note that for values that do not represent transfers, do not change through being read, or for which a Java-like PropertyChangeListener construct serves sufficiently (such as when modeling production rates instead of discrete product in a manufacturing environment), a simple field and event are more than adequate to communicate data between entites. In other words, unlike many block and port architectures, this is one of several first-class means you have at your disposal for communicating information between participants in your simulation.

12 MODEL ARCHITECTURE

HighMAST™ currently provides several model architectures implemented as prebuilt frameworks. All architectures are implemented on top of the HighMAST™ base engine and libraries, described above, and leverage some or all of their executive, resource, queue and other primitive functionalities. Some of the key architectures are Plant-Process-Product, Block-And-Connector, and Modular Supply Chain.

The Plant-Process-Product framework, depicted in **Figure 3**, below is built on the premise that there are three principal domains to some simulations – Plant, Process, and Product. Each has its own constructs and capabilities, and a wide range of simulations can be built up to any desired level of fidelity from primitives in each of these three domains.

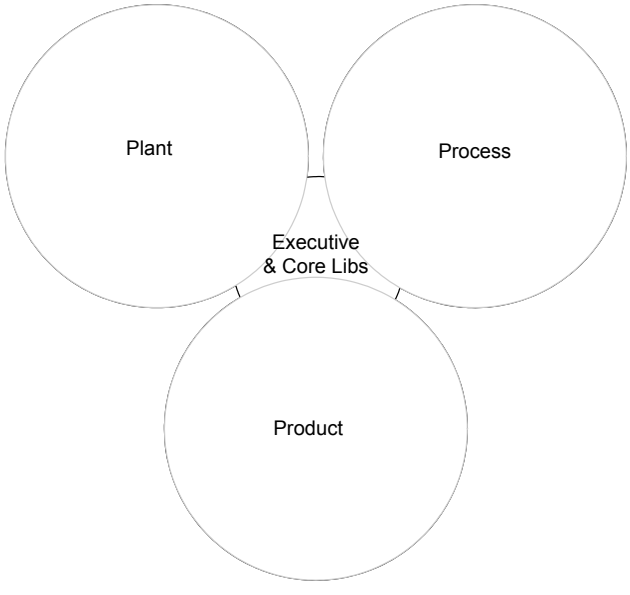

Figure 3: The Plant-Process-Product Framework

The Plant domain models that aspect of the real world where location and capability are critical. This domain incorporates elements for modeling plants, warehouses, equipment, vehicles, manufacturing stations, and other elements of an enterprise's Plant & Equipment assets. This is also the domain into which resource allocation, staff scheduling and cost analysis falls. Elements in the Plant domain are typically used by elements of the Process domain to manipulate elements of the Product domain. The plant domain is implemented primarily as resources and resource pools with allocation managers.

The Process domain models things that people and machines do. The workflow engine, described above, has been used to good effect, in describing and modeling complex procedures with interactions between manufacturing stations. A large task graph is built up, with one subpath for each of a number of manufacturing stations, and appropriate handoff and synchronization primitives to govern the progress of execution. Many of the tasks have resource acquisitions that draw from model-global, or manufacturing-station-local resource pools, as appropriate to the environment being modeled.

CPM and PERT analysis modules allow the analyst to gain a detailed understanding of the timing and dependencies of the model, or the enterprise application to make parameterized decisions in configuring exploratory simulations.

The Product domain is where a company's product is modeled. This domain utilizes a framework of items and transformations, shown in **Figure 4**, below, to model part-whole transformations of one set of items into one or more new items that have unique properties. Transformations may be bidirectional, and the new item may aggregate, retaining the identities of its constituent items, or transform, creating a new product. This enables modeling of a wide range of product transformations including chemical manufacturing, assembly of parts into components, subassemblies, assemblies and so forth, and boxing and palleting for shipment.

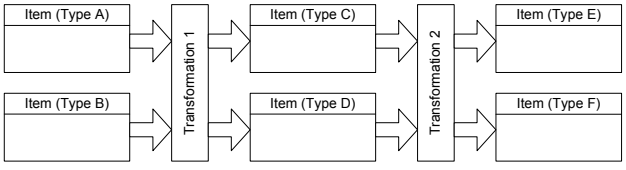

Figure 4: Item/Transformation Framework

Blocks and connectors are a well-known paradigm for building up models. HighMAST™ has a library of typical block constructs such as queues, distributions and item generators, and has been used to create a generic bank teller model and lake pollution model, which is available for inspection on the Highpoint Software website.

Modular Supply Chain - HighMAST™ has a modular supply chain modeling architecture which currently consists of a module called SupplyChainLink™. This module (N.B. This is still in development, but we expect it to be demonstrable before WSC '03) receives orders from above, maintains an inbound and an outbound inventory, receives shipments from below, and is capable of product transformation to represent palleting, manufacturing, transportation, etc, as described in Chopra (2000). We model customer pull through frequent small orders. The inbound inventory contains raw materials, and the outbound inventory contains finished goods. The SupplyChainLink class is capable of generating orders and transmitting them to adjacent links in the chain. Our SupplyChainMail™ permits a supply chain to incorporate many links at many levels to represent different service levels and product chains. Order & routing strategy objects exist at each level, and determine whether a link is push or pull, as well as providing replenishment levels, various cost data, and defining the individual chains. Separate channels in the link object can model material and financial cycles.

13 OTHER GOODIES

There are a few other tools, techniques and micro-architectures that have been developed for customers of Highpoint Software Systems as a part of our HighMAST™ efforts, that are powerful and useful enough to warrant mention here.

StatisticsLogger is an object that can be registered to listen for PropertyChangeEvents from any data that fires that event, and can aggregate statistics on the values and optionally the timing of the values taken on by that field.

Expressions – EvaluatorFactory is a class that can take a user-supplied string that represents an expression, and compile it, all while running in an application, into an executable module of code called an Evaluator, returning a function pointer (a delegate) to the compiled instance of that expression.

SmartPropertyBag is a dictionary (a set of name-value pairs, indexed and obtainable by using the name as a key) that can hold raw numeric data, delegates (function pointers) for dynamically determining the value to be returned for a key, expressions that may utilize other entries in the dictionary to dynamically determine the numeric value of a key, aliases to entries in other SmartPropertyBag constructs, and child (contained) SmartPropertyBag objects. Large networks of very flexible, user-programmable data entities (such as plant resources and heuristic strategies) can be modeled using these elements.

Mementos – Using a well-known design pattern from Gamma (1995), HighMAST™ includes a mechanism called a memento that we have used to enable a simulation participant to snapshot its state, and later reset that state from the snapshot or memento. The memento can represent an object hierarchy, and will be intelligent about how the snapshotting is performed. Elements that have changed since the last snapshot are re-recorded, and elements that have not changed, use the snapshot they recorded previously.

DbReflect is a framework that allows a nearly arbitrary object hierarchy to be accessible either as that object hierar-

chy, or as a set of tables, one-per-class, with a row per object. Tables may specify an index column, and changes to the table are instantly reflected in the underlying objects, and vice versa.

14 SUMMARY AND FURTHER READING

HighMAST™ is a new object oriented simulation framework, written in C#, that enables the designer to create simulations around a core event scheduling engine with a ports-and-connectors approach, a workflow-and-plant-model approach, or a range of other approaches. It exposes the full capabilities of Microsoft's .Net libraries to the designer, thereby enabling simpler web services, easier database and enterprise application integration, and better and more intuitive user interfaces created around WinForms or WebForms and animations written in DirectDraw.

HighMAST™ represents an acknowledgment that simulation development and application development are often one and the same, and takes a significant step toward bringing simulation development up to the state of the art in application development.

For further information on HighMAST™, please visit http://www.highpointsoftware.com.

REFERENCES

Chopra, S. et al. 2000. *Supply Chain Management.* ISBN 0-13-026465-2 Upper Saddle River, N.J. Prentice Hall

Cormen, T. et al. 2001. *Introduction to Algorithms.* ISBN 0-262-03293-7. MIT Press.

Gamma, E. et. al. 1995. *Design Patterns.* ISBN 0-201-63361-2. Reading, MA: Addison-Wesley.

Garrido, J. M. 1993. *Object-Oriented Discrete-Event Simulation with Java.* ISBN 0-306-46688-0. New York: Kluwer Academic / Plenum Publishers.

Law, A. M., and Kelton, W. D. 2000. *Simulation Modeling and Analysis*, 3rd ed. ISBN 0-070-59292-6 New York: McGraw-Hill.

Watkins, K. 1993. *Discrete Event Simulation in C*, ISBN 0-077-707733-4 England: McGraw-Hill Europe.

AUTHOR BIOGRAPHY

PETER C. BOSCH is a founder of Highpoint Software Systems, a small and attentive decision-support technology firm in the upper Midwest. He holds a BSEE from the State University of New York, and is a Certified Java Developer and Microsoft Certified Solution Developer. Pete has published numerous technical articles on object-oriented development in these environments. Pete has been designing and building simulations since 1991, for Fortune 100 firms in aerospace, medical imaging, pharmaceutical manufacturing and investment banking. He has been leading large software projects since 1995.