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

HDPS, AN XML/XSLT BASED HIERARCHAL MODELING SYSTEM

Richard Curry

London Business School
Regent's Park
London NW1 4SA, United Kingdom

Kiriakos Vlahos

Athens Laboratory of Business Adminsitration
Athinas Ave. & 2A Areos Str.
166 71 Vouliagmeni, Greece

ABSTRACT

HDPS is a practical system for designing modeling paradigms, creating hierarchal model definitions, and evaluating multi-paradigm models - particularly in business and finance. HDPS relies on XML (W3C 2004) to create model *types*, *definitions*, and *instances*. A *type* defines a modeling paradigm; for example, one *type* might define discrete event simulation while another may specify linear programming. A *definition* describes a system such as a firm's pricing decision process. A model *instance* is the state and history of a *definition* when operated upon by a *type*. Further, each *type* relies on one or more *implementations* to provide its functionality. xHDPS, a .NET version of HPDS, implements several modeling paradigms including simulation (discrete event, continuous time, and Monte-Carlo), optimization (linear and non-linear), knowledge-based expert systems, and general calculation (spreadsheet) models. A multi-generation service adoption model demonstrates a typical HDPS model structure with several interconnected models utilizing different modeling paradigms.

1 INTRODUCTION

Practitioners commonly employ multi-methodology problem solving and analysis (Munro and Mingers 2002). In many cases their work spans several paradigms, collections of methods and tools developed and used by different communities of researchers. These paradigms range from soft to hard modeling techniques and their models are combined using ad-hoc methods. Combining these models poses a difficulty because models in different paradigms often involve incompatible conceptualization procedures and require radically different structures and procedures. The challenge in creating these models lies in connecting model instances from dissimilar types and creating a unified modeling representation. Multi-paradigm models have been explored using many approaches. A common method employs multi-models(Ören 1991), and more recently this has extended to include meta-theoretic approaches for modeling the modeling process such as Traoré (2003). An example of a simulation-based model is the CAESAR simulation of complex adaptive supply chain networks by (Pathak, Dilts, and Biswas 2003).

There have been many research projects focusing on simulation and modeling for scientific and engineering systems, which have provided a good understanding of these areas. However, business and finance simulations typically face additional challenges due to of the lack of deep understanding about some of their aspects, such as the decision process of individuals. In addition, business systems tend to be loosely coupled and vary in structure as their systems evolve. The design and philosophy behind HDPS, the Hierarchal Dynamic multi-Paradigm System, reflects its primary purpose of modeling business and finance systems composed of interacting sub-systems.

Often in these systems, each sub-system may be best described using a modeling paradigm that is different from the ones required by others. HDPS addresses these by creating a flexible system for building hierarchal models with interchangeable sub-models built on multiple modeling paradigms. HDPS facilitates the model design process at three levels. The first simplifies the implementation and modification of modeling paradigms. The second creates a standard method of defining a model, regardless of which paradigm will be used to evaluate the model. The third allows the evaluation of a hierarchy of model instances within a unified modeling framework.

HDPS separates a model into four parts: *implementation*, *type*, *definition*, and *instance*. The *implementation* is the system which evaluates or simulates a model. Typically, the implementation is chosen due to the similarity and applicability of its paradigm to the system under evaluation. A *type* describes a cookbook for solving a model instance using a particular paradigm. For example a *type* can describe the transition rules for a queuing system. A *definition* is a description of a physical or logical system that abstracts objects in the system from their function. Finally, an *instance* is the state of the model while an *implementation* transforms its *definition* using the specified *type*.

In various types of modeling, these phases are combined in different manners. For example, the *implementation* and *type* are usually highly connected, particularly in methodologies based on software (i.e. a typical simulation package such as Arena or Extend). Abstracting the *definition* from the *type* is more common; for example, this has been done in optimization using structured modeling, a formalization of mathematical programming that takes the implementation away from model specification (Geoffrion 1987a) (Geoffrion 1987b). In the simulation area many papers have explored this subject including: Barros (1995),Barros, Zeigler, and Fishwick (1998), and Zeigler, Praehofer, and Kim (2000). Additionally, these techniques have been adopted for strategic modeling and industry level simulation (Ninios, Vlahos, and Bunn 1995).

While an HDPS model requires *implementations* that are able to execute all of the paradigms present in the multi-model hierarchy, each *definition* is not bound to any particular *implementation* because each *type* is not bound to a particular *implementation*. Likewise, it is possible for multiple types to use the same implementation. This is of particular benefit to those developing or extending modeling paradigms because for the same *type*, several *implementations* can be tested to determine which provides more satisfying answers. In the same manner, a *definition* may be used with more than one *type*. This is useful because some sub-systems may have several alternative methods for exploring their behavior and evaluating their performance. This occurs quite often because some economic systems have many theories for explaining and forecasting behavior but no established "laws" for doing so. Using several *types* on a *definition* allows the modeler to explore the behavior of the system under various theories to determine the robustness of the multi-model.

A typical HDPS model contains an evolving hierarchy of sub-models and acts as a multi-model. These might include an input / output model connected to a database, an outer model representing the state of the system, a model representing the transitions of the system, a model for the endogenous decisions in the systems, and a model representing the policy, the decision under the control of the modeler.

The practice of separating the *instance*, *definition*, *type*, and *implementation* in HDPS provides many benefits for a modeler. It clarifies an instance as one particular run of a model, which may be one of a number of identical or nearly identical models drawn from a common definition. The *definition* describes the system under study, separating the objective "what is" from the subjective "what will be," which results from a transformation contained in a *type*. The *type* specifies the paradigm used to generate model instances and as such logically represents changes that occur in a general class of models. An *implementation* is a program that creates an instance from a definition using a compatible *type*.

2 PARTS OF AN HDPS MODEL

An HDPS model has three parts: *type*, *definition*, and *instance*. The *type* defines a modeling paradigm by specifying the method of representation and transformations used upon *instances*. The *definition* is a logical description of the system being modeled. The *instance* is a running realization of the model consisting of the current state of the model, a set of inputs and outputs, and a list of the current transformation and any further transformations scheduled to occur.

2.1 Type

Types are reusable and extensible meta-models describing a modeling paradigm. In HDPS, *types* are composed of three parts: transformations, schema, and invariants. Transformations determine the actions (events) that a model of a given paradigm undergoes. The schema determines the structure of *definitions* as well as acting as a mechanism for verifying their contents. The invariants are lists modeling elements such as classes and objects that are present in all *instances* of the *type*.

An event, which may be scheduled by the *type*, *definition*, or the *implementation*, triggers the evaluation of a transformation. During a transformation, the *implementation* updates the *instance* according to rules specified in the *type*. The transformation rules in a *type* are set out as a functional program that acts on elements in an *instance*.

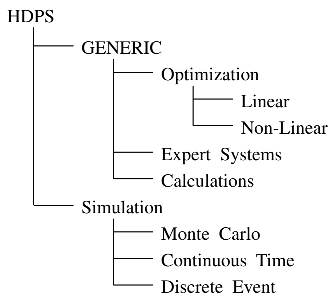
HDPS allows the extension of types through inheritance. Inheritance allows one model type to take functionality for a "parent" modeling type and extend it by adding or changing behavior. In HDPS, the inheritance mechanism allows the creation of type hierarchies composed of model *types*. The HDPS type hierarchy is rooted with HDPS *type*, which provides the basic functionality for communicating with the *implementation* and the multi-model *instance* hierarchy. Figure 1 shows an example type hierarchy for several common model types.

2.2 Definition

A model *definition* is a collection of elements that describes a portion of the system under consideration. Usually a definition provides a parameterization of a system, including the objects in the model and an interface that allows communication with other models. When used in conjunction with a compatible *type*, a *definition* creates an *instance* as shown in section 2.3.

In addition to functionality provided by the *type*, a definition can add functionality by creating or extending events. In these events, a definition specifies a custom

Figure 1: An Example HDPS Type Hierarchy



sequence of actions in addition to those present in the transformations defined by the *type*. Events do this by exposing a programming environment that allows them to interact directly with the *implementation* to expand the functionality of the basic type.

A *definition* can specify new classes and objects in addition to those provided by a *type*. As in object-oriented programming, an object represents a real world entity and a class defines its properties, methods, and events. Typically, a *type* defines most classes in the invariant section; however, HDPS allows extensible *types*, such as the GENERIC type, where *definitions* can specify additional classes. With in a class, properties are values associated with an object, methods are actions undertaken in response to a direct request, and events are actions taken in response to an event transformation. Although classes specify a default behavior for events, each object is able to override this functionality by specifying custom events. Although HPDS provides for a common standard for defining classes, object handling is *type* specific subject to the provision that the type follow the standard interfacing rules.

Building an HDPS multi-model has three steps: specifying a definition's interface, creating child instances, and linking instances. In HDPS, defining an interface requires specifying input and output ports for instances using *connectors* and *arguments*. Each *connector* exposes one internal object or property in an *instance* to other *instances*. *Connectors* are used in conjunction with *pipes*, which a definition uses to join connector either to their own or those in child models (see section 2.3). In addition to *connectors* and *pipes*, instances can pass input and output information through *arguments* at the beginning of each evaluation (and as such are not associated with events). While *connectors* and *pipes* provide persistent communications between models, arguments provide specific information for each call of the instance. For example, *arguments* could be used to pass input parameters, such as a particular sequence of events to call or implementation specific parameters, and output status.

2.3 Instance

An *instance* is a realization of a *definition* that is being transformed by a *type*. A single *instance* acts like a DEVS atomic model while a hierarchy of *instances* form a coupled model (Zeigler 1990). An *instance* has three primary components: state, event queue, and interface. The state describes the values for an instance at a particular instant; in particular, it is a set of objects that are persistent between transformations. The state can be conceptualized as a file containing all objects along with the current transformation index. During a typical transformation, the *implementation* loads the old state, changes it according to the rules specified in the *type*, and then saves the result as a new state.

The *instance* contains a list of pending events called the *event queue*. The event queue provides a schedule of transformations arranged by priority. While the queue priority is often considered to be time, it need not be as many paradigms do not include the notion of time. A prime example of this is optimization, which has many different steps without a notion of time. Events relate to many different types of simulation and modeling environments. Although the event queue was modeled like one used in discrete event simulation, it is also readily adaptable to other paradigms where events can correspond to stages in model setup and evaluation. For example, a mathematical program's events could preprocess the instance data, optimize the model, and post-process the output, or a continuous time simulation's event could update the system of differential equations at each Δt. The operation of each instance occurs according to a schedule as shown in Algorithm 2.3. Notice that the current state is not loaded or saved by default in an event as this is not considered a basic operation because the state will not be updated during each event. [ht] Instance Evaluation Load Input Arguments Select Next Event Evaluate Actions Schedule Dependent Events Next Event **is** *Empty* **or** Event Priority $>$ *Max Priority* Save Output Arguments

The final part of the *instance* is the interface. The interface controls child instances and the links between child models. During instance evaluation, a transformation can create, modify, evaluate, or drop a child *instance*. Linking these instances requires specifying *pipes*. A *pipe* contains all the associated properties, methods, and events for the *connectors* attached to the *pipe* as well as an index that indicates the last usage of the *pipe*. The *connector* on the receiving instance is responsible for determining if or how to use an object pushed on a *pipe*. By default a *connector* can use the first, last, min, max, average, or product of values on the *pipe(s)* for properties and can use the first or last for events. The index of a *pipe* is a stamp that defines when the object was pushed onto the *pipe*. The index allows a *instance* that attempts to pull the value from a pipe to check if it has a newer or older value than those on the *pipe*. When the values are pulled from a *pipe*, the model takes all of the values, unless the *pipe* is marked to preserve the data, in which case only its parent *instance* can free the data. A *pipe* can be connected to multiple *connectors* on many *instances*, and these connections can change during an *instance's* transformations. Using these mechanisms, HDPS supports dynamic structure modeling through dynamic model instancing and pipe re-routing.

3 USING XML AND XSLT

An HDPS model uses XML (eXtensible Markup Language) files to store all data, transformations, and meta-level information. Further, the standard tools that have grown around XML form the basic structure for implementations of HDPS. The use of XML and other markup languages is not new in modeling, and in particular there many examples of the use of XML technologies for simulation, most using it for model definition, data storage, and inter-model communication such as Fishwick (2002), and Vangheluwe and de Lara (2002). HDPS expands on these modeling efforts by using the inherent extensibility of XML to create a complete modeling environment that conforms to industry standards. Two key technologies used by HDPS are XML namespaces (W3C 1999a), which simplifies *type* inheritance, and the extensibility of XSLT (eXtensible Stylesheet Language Transformations) (W3C 1999b), which eases the development of new *implementations*. While hand-crafting XML is a tedious and difficult process, many tools exist for generating generic XML and the prevalence of XML has made it easy to extend these tools to create custom XML generators for HDPS *types*. This section continues by explaining how HDPS uses XML to specify *types*, *definitions*, and *instances*.

3.1 Types

A *type* defines a model paradigm. The *type* consists of a XSD schema that defines the available structures, an XSLT transformation that defines the operations that the model can undertake, and an XML invariant list, a file containing modeling elements for the initial *instance* and *state* that are not included in the *type*. For simple *types*, only the transformation file is necessary to evaluate a model. The schema is only necessary for verifying the XML in the definition and the invariant file is necessary only if the *type* contains standard classes and objects that are not defined in the transformation. XSLT directly supports much of the type inheritance by allowing a type transformation to load one or more existing HDPS type transformations using the import XSLT element and by adding additional invariants using the document function.

Event transformations use XSLT to connect the model *definition*, *instance*, and *implementation*. The functionality is implemented using XSLT functions or by functions in an embedded script or custom transformation engine. XSLT recommends adding namespaces for functions implemented in custom engines or embedded scripts. Reflecting this, the base HDPS type adds four new namespaces to implement its functionality, see Table 1 for the namespaces and their roles.

Table 1: Namespaces in HDPS *Type*

Namespace	Role
HDPS	Basic structures and actions
Model	Model and interface manipulation
System	Accesses the implementation
Stack	Accesses the operations stack

In custom *types*, the functions in these namespaces as well as functions in new namespaces define a modeling paradigm. Transformations call these functions for elements in the *definition* depending on the current event. Figure 2 provides part of the HDPS *type's* transformation file.

Figure 2: Template for HDPS:MODEL element

```
01 <xsl:template match="HDPS:MODEL">
02   <xsl:choose>
03     <xsl:when test="hdps:IsEvent('BUILD')">
04       <xsl:apply-templates select="HDPS:INTERFACE"
--          mode="BUILD"/>
05     </xsl:when>
06    <xsl:otherwise/>
07   </xsl:choose>
08   <xsl:apply-templates select="child::*[1]"/>
09   <xsl:apply-templates  mode="ACTION"
--         select="HDPS:EVENT[@NAME=hdps:CurrentEvent()]"/>
10   <xsl:apply-templates mode="TEST"
--         select="HDPS:EVENT" mode="TEST"/>
11 </xsl:template>
```

The transformation file shown in figure 2 handles the main evaluation loop that occurs on each evaluation of the model, except for startup of the first model in the multi-model. On the "BUILD" event (which is automatically scheduled on loading the model and tested for using the "IsEvent" function found on the hdps namespace), the interface is loaded, otherwise the model continues. To load the interface, XSLT performs a transformation on the HDPS:INTERFACE element that instructs the *implementation* to add the *definition's* connectors and arguments to the *instance*. Next, for every event, the first XML child object, which contains the specific *definition*, is transformed on line 07 according to its specific *type*. Then, the *type* will evaluate the definition specific events by calling a transformation on the specific event in the *definition*. Finally, the HDPS:MODEL transformation test all events in the definition to see if they should be added to the event queue.

3.2 Definitions

HDPS *definitions* use XML to specify the objects and structures in a model. Since a *type's* schema specifies the markup style for a *definition*, making a general description is difficult. However, each *definition* must conform to the HDPS *type* for its basic structure and interface.

All *definitions* reside in a project element, "HDPS:PROJECT", that forms the root element for a multi-model; however, not all definitions in a multi-models must reside in the same project – they can be included from other the project as well. Each *definition* in the model exists in an "HDPS:MODEL" element which contains the information about the preferred *type* and *implementation*. The first child element of HDPS:MODEL contains the objects and events for the specific *type*. Any further child elements of HDPS:MODEL represent the interface, which typically consists of a list of "HDPS:CONNECTOR" elements that define the connectors for an instance.

Figure 3 provides an example *definition* for a model of *type* GENERIC (a standard *type* that supports the core functionality for the HDPS namespaces). The *definition*, "Driver," contains one object, one connector, and one event. The object, "OBJ A," is a state variable of class "INTEGER" with a default value of **0**. On the "BUILD" event, this model instances a sub-model, "Child 1," of type "Calculation" using a specific transformation file and implementation. Then the event links "OBJ A" to connector "A" and connects "A" to "AA" on "Child 1" with "Pipe 1." Then, the model *Driver* schedules the "EVAL" event on "Child 1" and starts its execution.

In order to interoperate with existing techniques, XML based modeling languages, such as MathML (W3C 2003), can be embedded in an HDPS model. Typically, these would be integrated by having the respective *type* directly support the paradigm or by having the *type* call an external program that uses the embedded elements.

3.3 Instances

Three sets of XML elements define an HDPS *instance*: the state, event queue, and interface. Conceptually, these elements exist in an XML file appended to the *definition* during type transformations; however, none of the elements of the state need to be implemented as XML files as there may be large performance gains if the instance is held in some other format by the implementation. In the state, each object stores its current attribute values in a collection of XML elements. Each event on the event queue stores its name, priority, and, optionally, an identification number. The interface has the current list of child instances and pipe linkages. Figure 4 provides an example state file.

In addition to the state of individual instances, the multi-model contains XML elements representing the pipes

Figure 3: A Sample *Definition*

```
<HDPS:PROJECT>
  <HDPS:MODEL NAME="Driver" TYPE="GENERIC"
            TRANSFORM="GENERIC.XSLT">
    <GENERIC:MODEL xmlns:GENERIC="urn:GENERIC">
        <HDPS:EVENT NAME="BUILD">
           <HDPS:ACTIONS>
              <HDPS:LOAD_MODEL_DEF FILE="A.xml"
                        NAME="Child" TYPE="Calculation"
                        TRANSFORM="Calculation.XSLT"
                        IMPLEMENTATION="HDPSLib.dll"/>
              <HDPS:INSTANCE INSTANCE="Child 1"
                        DEFINITION="Child"/>
              <HDPS:LINK CONNECTOR="A" OBJECT="OBJ A"
                   ATTR="VALUE" DIRECTION="BOTH"/>
              <HDPS:ADD_PIPE NAME="PIPE 1"
                   FROM_CONNECTOR="A"
                   TO_MODEL="Child 1" TO_CONNECTOR="AA"/>
              <HDPS:EVALUATE NAME="Child 1" EVENT="EVAL"/>
           </HDPS:ACTIONS>
        </HDPS:EVENT>
        <GENERIC:OBJECTS>
          <GENERIC:OBJECT NAME="OBJ A" TYPE="INTEGER">
             <GENERIC:ATTR NAME="VALUE" STATE="TRUE"/>
             <GENERIC:DEFAULT ATTR="VALUE">
                <HDPS:VAL IS="0"/>
             </GENERIC:DEFAULT>
          </GENERIC:OBJECT>
        </GENERIC:OBJECTS>
    </GENERIC:MODEL>
    <HDPS:INTERFACE>
      <HDPS:CONNECTOR NAME="A" TYPE="INTEGER"
           MAX_CONNECTIONS="10" USE="LAST"/>
    </HDPS:INTERFACE>
  </HDPS:MODEL>
</HDPS:PROJECT>
```

that connect the hierarchy of instances. The implementation of the HDPS type handles the location and format of these files, which need not be implemented as XML files. However, when interfacing non-compatible implementations, the standard format for both arguments and pipes consists of an indexed list of objects and attribute values. For example, figure 5 shows that an "INTEGER" typed pipe, "Pipe 1," has a property, "VALUE," of 10 at index 1.

4 XHDPS, A .NET IMPLEMENTATION OF HDPS

xHDPS is an implementation of HDPS built upon the Microsoft .NET technologies ©(Microsoft 2004). In particular, xHDPS implements its core libraries as a C++.NET assembly and provides a generic interface for models written in C#. xHDPS contains a working set of the core HDPS type as well an implementation of several types representing modeling paradigms such as continuous time simulation (systems of differential equations), linear and non-linear optimization, backward and forward chaining knowledge based systems, and general calculations for spreadsheet modeling techniques.

In addition to the purely modeling types shown in figure 1 and mentioned above, xHDPS contains types to support user interfaces and databases. Of these types the HDPS

Figure 4: Initial State for the Sample *Definition*

```
<STATE:INSTANCE NAME="A" DEFINITION="DEF"
     TYPE="GENERIC">
  <STATE:CURRENT_INDEX IS="1">
  <STATE:OBJECTS>
    <STATE:OBJECT NAME="A" TYPE="INTEGER">
      <STATE:INDEX>1</STATE:INDEX>
      <STATE:ATTR NAME="VALUE">10</STATE:ATTR>
    </STATE:OBJECT>
  </STATE:OBJECTS>
  <STATE:EVENT_QUEUE>
    <STATE:EVENT NAME="BUILD" PRIORITY="-1" ID="1"/>
  </STATE:EVENT_QUEUE>
  <STATE:INTERFACE>
    <STATE:PIPES>
      <STATE:PIPE NAME="PIPE 1" LOCATION="PIPE 1"
           FROM_CONNECTOR="A"
           TO_INSTANCE="CHILD 1" TO_CONNECTOR="AA"/>
    </STATE:PIPES>
    <STATE:SUBMODELS>
      <STATE:SUBMODEL  NAME="CHILD 1"
           DEFINITION="CHILD MODEL TYPE="Cacluation"
           TRANSFORM="CalculationEngine.XSLT"
           IMPLEMENTATION="HDPSLib.dll"
  </STATE:INTEFACE>
</STATE:PROJECT>
```

Figure 5: A Pipe from the Sample *Definition*

```
<PIPE:PIPE NAME="PIPE 1" TYPE="INTEGER"
     PRESERVE="YES" PARENT="DRIVER">
  <PIPE:INDEX IS="1">
    <PIPE:ATTR NAME="VALUE">10</PIPE:ATTR>
  </PIPE:INDEX>
</PIPE:PIPE>
```

core library supports a database model, DATAIO, and a simple user interface library, HDPS_UI. Beyond the core library, xHDPS implements other *types* as .NET assemblies that are loaded at run time. In this manner, xHDPS offers co-simulation through the inclusion of different *implementations* in a multi-model.

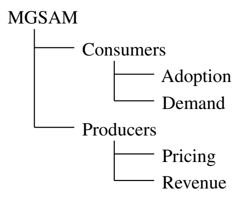
Additionally, .NET allows for great flexibility in the design of new model types and implementations. Model types and implementations are contained in .NET assemblies derived from a .NET class that implements the HDPS *type*. To create these assemblies, .NET facilitates the extension of the XSLT transformation engine by assigning the functions of .NET classes to XSLT namespaces. Further, because the .NET framework is language neutral, new *implementations* can be added in any programming language supported by .NET. Additionally, existing code may be wrapped with custom classes derived from the base class, HDPSModel, using C++.NET. In xHDPS, *implementations* of types are added to the multi-model by loading their associated .NET assemblies into the instances of the multi-model.

In xHDPS, an *implementation* can be tailored to a *definition* and *type* to improve performance over a more generic approach by overriding the functionality of the parent *type* or *implementation*. For example, the traveling salesperson problem (TSP) can be implemented as a generic integer program or it could be solved using problem specific heuristics that can provide a significant performance boost.

5 MULTI-GENERATION SERVICE ADOPTION MODEL

This section presents a model of the adoption of a multi-generation service that is provided by a profit-maximizing monopolist. The monopolist initially sell a service, A, and at some later date introduce a new service, B, that is at least as good in every respect, except price, as A. At each period, the monopolist will optimize its profits by selecting prices for each service. Given this price, consumers decide what if any product they should adopt. A hierarchal model of this system contains six models: MGSAM, Consumer, Adoption, Demand, Producer, Pricing, and Revenue. Figure 6 depicts this multi-model as a hierarchy or system entity structure of the multi-model.

Figure 6: Instance Hierarchy for the Multi-Generation Service Adoption Model



The top-level model, MGSAM, controls the input/output and coordination of the consumer and producer sub-models. The consumer model coordinates the operations of its two sub-models, the adoption and demand model. The adoption model determines the number of consumers using each service with the differential equations 1 through 4 (Norton and Bass 1987):

$$S_1(t) = F_1(t) \cdot m_1 \qquad t \leq \tau_2 \qquad (1)$$

$$S_2(t) = 0 \qquad t \leq \tau_2 \qquad (2)$$

$$S_1(t) = F_1(t) \cdot m_1 \cdot [1 - F_2(t - \tau_2)] \qquad t > \tau_2 \qquad (3)$$

$$S_2(t) = F_2(t - \tau_2) \cdot [m_2 + F_1(t) \cdot m_1] \qquad t > \tau_2 \qquad (4)$$

Given that $S_i(t)$ is the market share for generation i at time t, m_i is number of adopters of the i^{th} generation. τ_2 is the time of the introduction of the second product. P_i is the innovation coefficient and Q_i is the imitation coefficient. These can be unrolled into a series of differential equations that can be simulated using a methodology similar to that employed by system dynamics.

In a typical diffusion model, the fraction of customers adopting the service at time t, $F_i(t)$, is shown in equation 5. In our multi-model, this value is replaced by a price-based demand function contained in the demand model.

$$F_i(t) = \frac{1 - e^{-(Q_i + P_i)\cdot t}}{1 + \frac{Q_i}{P_i} \cdot e^{-(Q_i + P_i)\cdot t}} \tag{5}$$

The demand model determines the demand based on a multi-stage demand equation (6), and the price that the monopolist determines. This demand function replaces the fractional demand shown in equation (5) (for more information see (Curry and Vlahos 2003)).

$$d_{i,t} = M_{i,t}(Q_{i,t}) \cdot \left(1 - \frac{p_{i,t}}{\Pi_{i,t}(P_{i,t})}\right) \tag{6}$$

The producer model complements the consumer model by modeling the pricing strategy of the producer and the revenue of the monopolist. In the pricing sub-model, a monopolist optimizes its pricing strategy at each point in time as shown in equation (7). While this is actually a somewhat poor policy compared to a multi-period optimization, it accurately represents many firms' decisions.

$$\max \pi = \sum_{t=1}^{n} \left[(p_i(t) = c_i \cdot (X(x, p, t))) \cdot x_i(x, p, t) \right] \tag{7}$$

The revenue model determines the actual revenue of the firm by integrating the price, p_i the number of customers, x_i, for each technology as shown in (8).

$$Revenue = \int_0^t \sum_i x_i \cdot p_i \tag{8}$$

The output of the model is the adoption curve for both generations and the total revenue of the monopolist. In practice, this model could be used by varying the parameters in the demand model to see the effect different demands have on the adoption of each generation. In these scenarios, a policy model could be added to test a set of regulatory of industry policies in order to maximize the utility of the system. Finally, the structure of the multi-model allows it to be expanded to include factors such as investment, capacity, competition, and government regulations as done in Curry (2004).

6 CONCLUSION

The desirable and increasingly common practice of building models that combine many modeling methodologies has been encumbered by the complexity of integrating models from different paradigms, particularly because of differences in their conceptualization and definition. HDPS was developed to simplify this process by adopting a standards-based representation for modeling. The core idea of HDPS is to decouple the modeling process into four design stages: *type*, *definition*, *instance*, and *implementation*. An HDPS *type* provides a common language for describing paradigms, which simplifies development and extension of modeling techniques. A system of *definitions* provides the ability to create multi-models by plugging in component *instances* developed in many different paradigms. Further, HPDS allows a modeler to experiment with *definitions* using different *types* and *implementations*. HDPS handles the inherent complexity of multi-paradigm modeling by adopting XML and other industry-standard tools to create a paradigm-neutral representation. In summary, modelers using HDPS can build extensive multi-paradigm models without having to resort to ad-hoc methods, which allows them to build increasingly complex models of financial and business systems.

REFERENCES

Barros, F. J., B. P. Zeigler, and P. A. Fishwick. 1998. Multimodels and dynamic structure models: an integration of dsde/devs and oopm. In *Proceedings of the 1998 Winter Simulation Conference*, ed. D. J. Medeiros, E. F. Watson, J. S. Carson, and M. S. Manivannan, 413–419. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Barros, F. J. 1995. Dynamic structure discrete event system specification: A new formalism for dynamic structure modeling and simulation. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos and K. Kang. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Curry, R. E., and K. Vlahos. 2003. The effect of regulatory regimes and market structure on consumer adoption of broadband. In *Proceedings of the ITS Europe Conference*. ITS Europe.

Curry, R. E. 2004, Forthcoming. *Hierarchical multi-paradigm modeling: A method for formalizing and evaluating dynamic economic systems, with an application to the adoption of consumer broadband*. Ph. D. thesis, University of London.

Fishwick, P. A. 2002. Using xml for simulation modeling. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yucesan, C.-H. Chen, J. Snowdon, and J. Charnes, 616–622. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Geoffrion, A. M. 1987a, January-February. The formal aspects of structured modeling. *Operations Research* 37:30–51.

Geoffrion, A. M. 1987b, May. An introduction to structured modeling. *Management Science* 33:547–588.

Microsoft 2004. Microsoft .net framework home. Available online via <http://msdn.microsoft.com/netframework/>. [accessed March 21, 2004].

Munro, I., and J. Mingers. 2002. The use of multimethodology in practice – results of a survey of practitioners. *Journal of the Operational Research Society* 53:369–378.

Ninios, P., K. Vlahos, and D. W. Bunn. 1995. Oo/devs: A platform for industry simulation and strategic modelling. *Decision Support Systems* 15:229–245.

Norton, J. A., and F. M. Bass. 1987, September. A diffusion theory model of adoption and substitution for successive generations of high-technology products. *Management Science* 33 (9): 1069–1086.

Ören, T. I. 1991. Dynamic templates and semantic rules for simulation advisors and certifiers. In *Knowledge Based Simulation: Methodology and Application*, ed. P. A. Fishwick and R. B. Modjeski. New York: Springer Verlag.

Pathak, S. D., D. M. Dilts, and G. Biswas. 2003. A multiparadigm simulator for simulating complex adaptive supply chain networks. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, P. Sánchez, D. Ferrin, and D. J. Morrice, 808–816. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Traoré, M. K. 2003. A meta-theoretic approach to modeling and simulation. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, P. Sánchez, D. Ferrin, and D. J. Morrice, 604–612. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Vangheluwe, H., and J. de Lara. 2002. Meta-models are models too. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yucesan, C.-H. Chen, J. Snowdon, and J. Charnes, 597–605. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

W3C 1999a, January. Namespaces in xml. Available online at <http://www.w3.org/TR/REC-xml-names>. [accessed March 21, 2004].

W3C 1999b, November. Xsl transformations (xslt). Available online at <http://www.w3.org/TR/1999/REC-xslt-19991116>. [accessed March 21, 2004].

W3C 2003, October. Mathematical markup language. Available online at <http://www.w3.org/TR/2003/REC-MathML2-20031021/>. [accessed March 21 2004].

W3C 2004, February. Extensible markup language (xml) 1.0 (third edition). Available online at <http://www.w3.org/TR/2004/REC-xml-20040204>. [accessed March 21, 2004].

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*. 2nd ed. Orlando, FL, USA: Academic Press.

Zeigler, B. P. 1990. *Object oriented simulation with hierarchical modular models*. Academic Press.

AUTHOR BIOGRAPHIES

RICHARD E. CURRY is a Ph.D. candidate at the London Business School. His research develops methodologies suitable for modeling complex business and economic situations, particularly for policy analysis. The major application area for his work has been telecommunications regulation. His e-mail address is <rcurry@london.edu>.

KIRIAKOS VLAHOS is an Associate Professor of Decision Science at the Athens Laboratory of Business Administration (ALBA). His early research was relying on the use of large-scale optimization models, but he is now investigating the use of more flexible decision support frameworks that allow the integration of ”hard” and ”soft” approaches. The main application areas for his work have been the study of competitive energy and telecommunications markets. He has published articles in the European Journal of Operational Research, the Journal of the Operational Research Society, Decision Support Methods, Energy Economics, Technological Forecasting and Social Change and Fiscal Studies. He has edited a book on Decision Science and special academic journal issues, and has organized two international conferences. His e-mail address is <kvlahos@alba.edu.gr>