Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

FLEXIBLE INTEGRATION OF XML INTO MODELING AND SIMULATION SYSTEMS

Mathias Röhl
Adelinde M. Uhrmacher

University of Rostock
Albert-Einstein-Str. 21
Rostock, 18059 GERMANY

ABSTRACT

As the effort towards standardization of formalism representations increases so does the need for verifying whether models do or do not follow a standard. Data binding allows to systematically exploit XML and its associated technologies for modeling and simulation purposes. Based on the schema definition of a formalism, a binding compiler generates model classes that support the user in constructing models according to the formalism. Most constraints can be checked automatically, few require separate efforts by the designer of the simulation system. Although simulators could be build for these declarative model descriptions, they would be hardly efficient. To this end, a separate transformation component is required. In this overall process, both model specifications that are consistent with a formalism definition and models that can be executed efficiently are supported equally.

1 INTRODUCTION

Interoperability of modeling and simulation tools can be largely increased by adherence to standard exchange formats that allow the in- and export of models specified in certain modeling formalisms. Such formats exist for Petri nets (Billington et al. 2003) and hybrid systems (MoBIES 2004). For discrete event system specifications, only suggestions exist so far, e.g. (Wang and Lu 2002, Hong, Kim, and Kwon 2000, Schäfer 2004), but non of them has been established as a standard yet. Currently, the *DEVS Standardization Group* of the *Simulation Interoperability Standards Organization* promotes the development of a standard format for DEVS (SISO 2005).

XML is expected to play a central role for model exchange (Tolk 2004). Already now XML is widely used in modeling and simulation, e.g. in combination with DTD (Filippi and Bisgambiglia 2002) and DOM (Fishwick 2002). However, the capabilities of XML are often not fully exploited and unnecessarily much effort is spent in implementing XML handling.

The paper is structured as follows. We will first give a short introduction into data binding and compare it to document centric approaches like DOM, which has been used widely for modeling and simulation. We will shortly sketch how systematical use of data binding supports the construction of syntactically correct models that adhere to individual model formalisms and established or still evolving "standards". Recommended extensions refer to a) checking for further constraints and b) transforming the generated models into efficiently executable models. Thereafter we will present how this approach is utilized in James.

2 DATA BINDING FOR A VALID AND EFFICIENT MODELING

XML (eXtensible Markup Language) has become the quasi-standard for storing semi-structured data. Generally, data handling based on XML is robust, extensible, and adopts easily to complex structures (Harold 2002). XML documents have a tree structure that is made up of nested elements. The classic approach to access XML data is via Document Object Model (DOM) or Simple API for XML (SAX). DOM is document centred. The whole XML-document is transformed to a tree of objects reflecting the structure and holding the content of the document's data. The data tree can be accessed and edited at will. In contrast to that SAX is an event based. It successively produces events each reflecting the node at hand of the XML document. The application is responsible for building object structures in memory.

Both DOM and SAX allow to map between XML documents and objects that can be used within software. DOM and SAX work on a rather low level and are document centered. They provide access to entities that directly reflect XML's structure: elements and attributes. One problem with this is that type safety is lost. Furthermore, decoding and encoding from and to XML has to be done manually. One usually wants to work on entities that reflect the structure of the application domain rather than that of the underlying storage technology. E.g. in the case of modeling and simulation ports, couplings, and models should be the focus of interest. This is called a data centric approach (McLaughlin 2002).

Schema languages impose well-defined constraints XML data can be checked against. Document Type Definitions (DTD) define occurrence and ordering constraints on elements and whether required attributes are present. In addition to DTD's features, XML Schema Definitions (XSD) allow to constrain the contents of elements and attributes by type and value range assignments. Standardized by the W3C XSD is written itself in XML and supports structuring of definitions by means of namespaces.

With DOM or SAX validation of XML data has to be done on the entire XML document. An alternative is to build Java representations of schema constraints and apply these to data (McLaughlin 2000). This supports an efficient and well scaling validation of data based on schema definitions. Today's data binding frameworks take a similar approach with the additional benefit of providing generation techniques to minimize manual coding.

2.1 XML Data Binding

Data binding is a powerful approach to implement XML support within software systems (Brodkin 2001, Bourret 2005). Given a set of schema definitions a compiler generates source code that enforces data to adhere to the constraints. Thus, a data binding framework makes manual encoding of internal representations and consequently its transformation to and from XML data unnecessary. The generated source code can directly be used for representing model data and it provides automatic conversion to and from XML.

Advantages of data binding are easiness of updating format specifications and type security of XML handling. Whereas DOM or SAX recognize errors, e.g. whether certain elements or attributes are present, only at runtime, in data binding inconsistencies are recognized by the compiler of the target programming language.

Data binding frameworks exist for different target languages, e.g. Java (Ort and Mehta 2003) and C++ (Ware 2005). In the following, we will use JAXB which stands for Java Architecture for XML Binding (Sun 2005).

2.2 Modeling Based on Data Binding

Data binding is well suited for the area of modeling. It has the potential to make proprietary data formats for model representations superfluous. With the help of data binding the use of standard formats is not limited to the exchange of models. If standards exist, they can directly be put to use within modeling and simulation tools. In addition it supports a low cost adaptation if standards — as it is the case with DEVS — are still developing.

For modeling and simulation systems data binding eases the construction of models and their transformation. Figure 1 gives an overview of the different representations of models and formalisms and activities of users supported by data binding. We will shortly describe the successive steps:

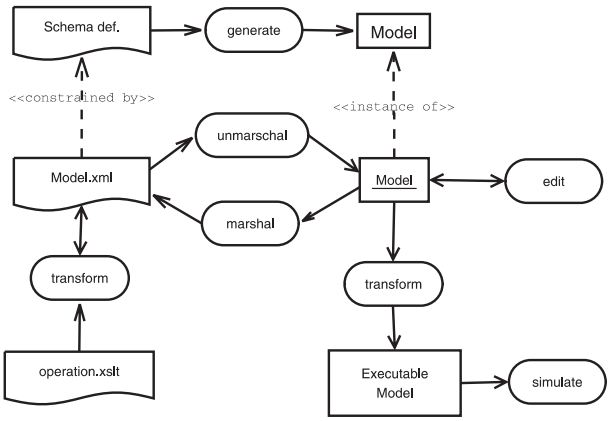

Figure 1: XML Integration Based on Data Binding

- A model formalism is defined or an existing standard specification is used as a schema definition (e.g. Document type definition, XML Schema Definition, or RelaxNG).
- A binding compiler generates source code classes that encour the schema based on the schema definition.
- Given these classes and maybe additionally defined constraints, the user is able to edit and design "correct" models.
- The thus constructed models can be stored via marshalling in XML. Conversely, unmarshalling transforms XML data to objects.
- XML descriptions of models can be manipulated via XSLT.
- Object representations of models can be transformed into efficiently executable model objects, e.g. by replacing declarative parts with object references. These models form the base for the simulation engine.

Within *generation*, customization files can be used for tool specific configurations, e.g. names for target packages, classes, and methods. This step has to be done once for a modeling formalism and for each change in the specification of a formalism respectively.

Marshalling takes content objects and converts them to XML data. *Unmarshalling* takes XML model data and instantiates the generated classes into a tree of Java content objects according to the XML data. Unmarshalling checks syntactical correctness, i.e. it validates the models against the original schema definitions. Marshalling and Unmarshalling are built-in capabilities of the generated classes.

Using data binding ensures that constructing a model results in a model that conforms to the according schema definition. The content objects can be validated on-demand during *edition*.

Working with XML representations of models two types of *transformations* are considered in our approach. All technologies that directly work on XML can be easily integrated. Thus, the framework is open with respect to models that are transformed into supported modeling formalisms via XSLT (W3C 1999). In contrast to that, XML representations of models not the best choice model execution. To simulate efficiently a *transformation* of the content objects into an execution optimized representation becomes necessary. Of course, the executable representation has to be reflected by according simulation engines.

Let us take a look at a simple example. We use XML Schema definitions for the PDEVS formalism. However, the focus is on how schema representations of formalisms can be exploited for modeling and simulation tools. The approach works analogous for other formalisms and schema languages respectively.

Example 1 *Extract of XML Schema Definition for PDEVS models:*

```
<xsd:complexType name="Model" abstract="true">
 <xsd:attribute name="name" type="xsd:string"
                use="required"/>
</xsd:complexType>

<xsd:complexType name="BasicDEVS" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="Model">
   <xsd:sequence>
    <xsd:element name="inport" type="Port"
         minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="outport" type="Port"
         minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="BasicCoupledDEVS"
     abstract="true">
 <xsd:complexContent>
  <xsd:extension base="BasicDEVS">
   <xsd:sequence>
    <xsd:element name="modelname" type="xsd:string"
         minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="eic" type="Coupling"
         minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="ic" type="Coupling"
         minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="eoc" type="Coupling"
        minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="CoupledPDEVS">
 <xsd:complexContent>
  <xsd:extension base="BasicCoupledDEVS">
   <xsd:sequence>
    <xsd:element name="model" type="BasicDEVS"
         minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="BasicAtomicDEVS"
     abstract="true">
 <xsd:complexContent>
  <xsd:extension base="BasicDEVS">
   <xsd:sequence>
    <xsd:element name="state" type="State"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="AtomicPDEVS">
 <xsd:complexContent>
  <xsd:extension base="BasicAtomicDEVS">
   <xsd:sequence>
    <xsd:element name="deltaInt"
                 type="xsd:string"/>
    <xsd:element name="deltaExt"
                 type="xsd:string"/>
    <xsd:element name="deltaCon"
                 type="xsd:string"/>
    <xsd:element name="lambda"
                 type="xsd:string"/>
    <xsd:element name="timeAdvance"
                 type="xsd:string"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Within the XML Schema Definition all model types are derived from an abstract complex type *Model* with a single name attribute that is required for a valid *Model*. *BasicDEVS* defines the interface for DEVS models based on input and output ports. It forms the base for all DEVS models. *BasicAtomicDEVS* and *BasicCoupledDEVS* derive from *BasicDEVS* and add elements according to atomic and coupled DEVS models. *AtomicPDEVS* and *CoupledPDEVS* extend the basic DEVS types to non-abstract types according to the parallel DEVS formalism. At this stage, it is only important to have a schema definition of some kind no matter whether it is the optimal one.

Given the above schema definition JAXB 2.0 EA (Java.net 2005) generates the Java classes depicted in Figure 2. The model classes reflect the schema structure directly. Simple XML Schema types like strings are mapped to according Java types. Complex XSD types are mapped to Java classes preserving dependencies and inheritance relationships.

Instances of these classes can now be used to construct and edit models. Marshalling of such a model will produce XML data, e.g. Example 2, that validates against the Schema definition given in Example 1.

Example 2 *XML-specified model of a grid element according to the Schema definition of Example 1.*

```
<model xsi:type="CoupledPDEVS" name="GridElement">
  <inport type="FireParcel">InNorth</inport>
  <inport type="FireParcel">InSouth</inport>
  <inport type="FireParcel">InWest</inport>
  <inport type="FireParcel">InEast</inport>
  <outport type="FireParcel">OutNorth</outport>
  <outport type="FireParcel">OutSouth</outport>
  <outport type="FireParcel">OutWest</outport>
  <outport type="FireParcel">OutEast</outport>
  <submodelname>FireModule<submodelname>
  <eic fromModel="this" toPort="InNorth"
       fromPort="InNorth" toModel="Fire"/>
  <eic fromModel="this" toPort="InWest"
       fromPort="InWest" toModel="Fire"/>
  <eic fromModel="this" toPort="InEast"
```

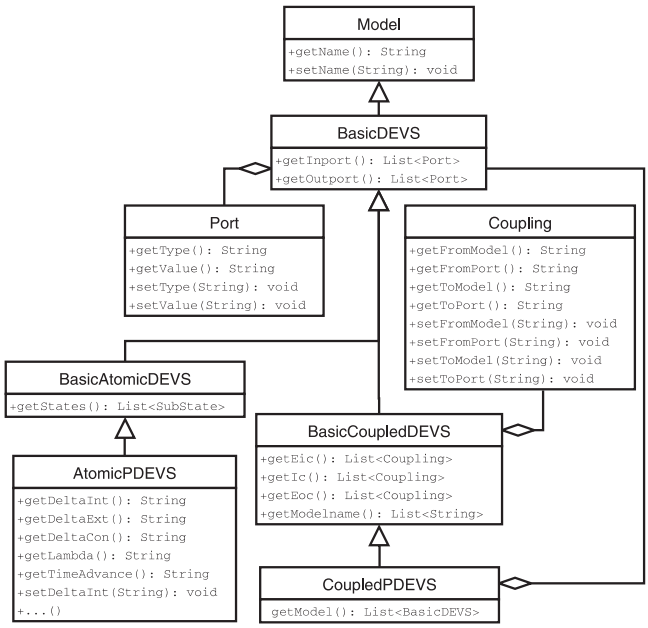

Figure 2: Classes Generated From The PDEVS Schema Definition

```
      fromPort="InEast" toModel="Fire"/>
  <eic fromModel="this" toPort="InSouth"
       fromPort="InSouth" toModel="Fire"/>
  <eoc fromModel="Fire" toPort="OutNorth"
       fromPort="OutNorth" toModel="this"/>
  <eoc fromModel="Fire" toPort="OutSouth"
       fromPort="OutSouth" toModel="this"/>
  <eoc fromModel="Fire" toPort="OutWest"
       fromPort="OutWest" toModel="this"/>
  <eoc fromModel="Fire" toPort="OutEast"
       fromPort="OutEast" toModel="this"/>
  <submodel xsi:type="AtomicPDEVS" name="Fire">
    ...
    <state>
      <substate type="java.lang.String"
        init="null">fullName</substate>
      <substate type="java.lang.Integer"
        init="0">phase</substate>
      <substate type="java.lang.Boolean"
        init="true">SpreadToNorth</substate>
      <substate type="java.lang.Boolean"
        init="true">SpreadToEast</substate>
      <substate type="java.lang.Boolean"
        init="true">SpreadToSouth</substate>
      <substate type="java.lang.Boolean"
        init="true">SpreadToWest</substate>
    </state>
    <deltaInt>...</deltaInt>
    ...
  </submodel>
</model>
```

3 EXTENSIONS

Generated model classes could be used directly for constructing models that adhere to the according schema definition without bothering about XML specifics like elements and attributes. Furthermore, they can be used for other activities like visualization. However, data binding cannot solve all problems relevant for model construction. There are likely to be cases where one does not want to refrain from additional manual coding.

By a binding framework generated classes provide a solid ground for constructing models according to a certain schema. But, sometimes the schema language of choice might not be able to express everything that needs to be expressed or the binding compiler might not fully support the schema language. For example JAXB 1 does not support type substitution and uniqueness constraints.

In the following we demonstrate the need for extended constraint checking at the example of the classes generated in the prior section. We will compare possible solutions to implement additional constraint checking and finally provide a solution based on a bunch of editor classes.

3.1 Additional Constraint Checking

Some constraints are not or difficult to express with XSD. For example XML Schema definitions allow only basic constraints between sets of elements. For constructing valid model instances additional constraints tests are required very likely.

Let us take a look at *CoupledPDEVS*. Within the generated class the getter for sub models returns a list of *BasicDEVS* models. If we derive atomic and coupled models for sequential DEVS from *BasicDEVS* we could also add *AtomicDEVS* models to the list of sub models (in addition to *AtomicPDEVS* and *CoupledPDEVS*). This is not in conformance with the PDEVS formalism, where coupled models should only accept PDEVS models as sub models. What we would actually have expected here is a list of *BasicPDEVS* models. XML schema definitions are not able to provide this type safety for sub models due to the lack of multiple inheritance. Because in our example all DEVS models already extend *BasicDEVS* we cannot introduce a type *PDEVS* and let *AtomicPDEVS* and *CoupledPDEVS* extend it as well.

To change the code of *CoupledPDEVS* manually is not really a solution. A change of the according schema and the subsequently done generation of the Java class files would overwrite the changes and require manual coding efforts. Thereby a main benefit of the generative approach would be lost. To come around we need an additional class to implement the desired feature.

The most obvious solution would be to derive from CoupledPDEVS and implement *addModel()* functions that take appropriate arguments, i.e. *AtomicPDEVS* and *CoupledPDEVS*. But since Java does not support multiple class inheritance, we would loose flexibility in implementing the editor classes.

A better solution is to create a further tree of classes that corresponds to the class hierarchy of the generated classes. Figure 3 shows editors for the generated classes representing parallel DEVS models. Each editor is responsible for one of the generated model classes and implements only function-

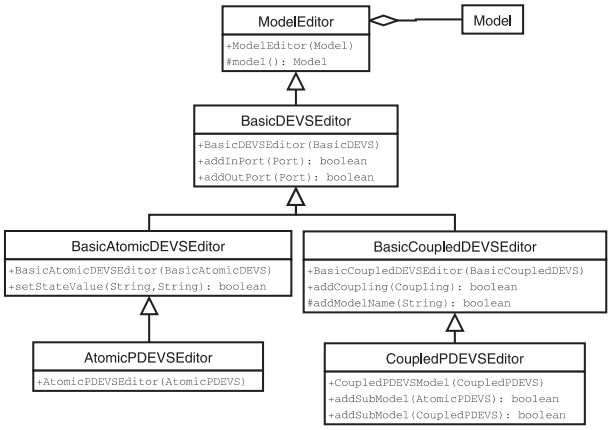

Figure 3: Editors for Bound PDEVS Models

ality not already covered by the according schema definition. For example *CoupledPDEVSEditor* operates on *CoupledPDEVS* and takes *AtomicPDEVS* and *CoupledPDEVS* as arguments for sub model addition.

There are more problems with our generated classes. When constructing coupled PDEVS models we would like to ensure coincidence of sub model references (a set of string values) with sub models (set of *BasicDEVS* models) actually contained. Within XML Schema Definitions this could actually be modeled by creating a separated type for a submodel and enforce the occurrence of both the name and the actual model. But, this would change the structure of our formalism representation. That is, the expressiveness constraints of the schema language would introduce artifacts into the model types. Instead, *CoupledPDEVSEditor* checks this within the two *addSubModel()* methods.

Furthermore, for coupled PDEVS models we would like to ensure that couplings are specified on existing sub models. Example 3 shows how this is done by *BasicCoupledDEVSEditor*.

Example 3 *Java code that is used within* `BasicCoupledDEVSEditor` *to check the validity of couplings to add:*

```
public boolean addCoupling(Coupling coupling) {
  BasicCoupledDEVS model =
    (BasicCoupledDEVS) model();
  List<String> nameList = model.getModelname();
  String fromModelName = coupling.getFromModel();
  String toModelName = coupling.getToModel();

  List<Coupling> couplingList = null;
  if (nameList.contains(fromModelName)
      && nameList.contains(toModelName)
      && !fromModelName.equals(toModelName)) {
    couplingList = model.getIc();
  }
  else if (fromModelName.equals("this")) {
    couplingList = model.getEic();
  }
  else if (toModelName.equals("this")) {
    couplingList = model.getEoc();
  }
  else {
    return false;
  }
  couplingList.add(coupling);
  return true;
}
```

3.2 Description Elements

Sometimes schema definitions are not sufficiently restrictive, but for some cases there is simply no schema definition available. The latter is the case for the behavioral part of atomic PDEVS models. Transition functions of atomic PDEVS models could be arbitrary functions. However, for specifying complete simulation models within XML, we need description elements for atomic models, too.

A solution is the use of a programming language. Thereby, we are able to declare methods and require the XML data of an according type to provide the body of these methods. Figure 4 shows the definition of an interface for atomic PDEVS models. Now we can fill the *deltaInt*, *deltaExt* etc. slots of *AtomicPDEVS* with Java code according the *IPDEVS* interface. The drawback is that most of benefits of XML are lost when falling back to unstructured data. Nevertheless, specification of atomic model behavior become at least possible. They can be done according to Example 4, which shows the implementation of the time advance function of an atomic model.

Example 4 *Java implementation of the internal transition function within an atomic PDEVS model:*

```
...
<deltaInt>
  Integer phase = (Integer) s.getValue("phase");

  if (phase.equals("prepare to smoulder")) {
    s.setValue("phase", "smouldering");
    return;
  }
  else if (phase.equals("smouldering")) {
    s.setValue("SpreadToNorth", true);
    s.setValue("SpreadToEast",  true);
    s.setValue("SpreadToSouth", true);
    s.setValue("SpreadToWest",  true);

    s.setValue("phase", "burning");
    return;
  }
  else if (phase.equals("burning")) {
    s.setValue("phase", "inferno");
  }
  else if (phase.equals("inferno")) {
    s.setValue("phase", "burnt out");
    return;
  }
  else {
    s.setValue("phase", "inactive");
    return;
  }
</deltaInt>
...
```

For executing such models we have to assemble a class that implements *IPDEVS* according to the method bodies defined within the elements of *AtomicPDEVS*. Finally, the class has to be compiled before ready to use.

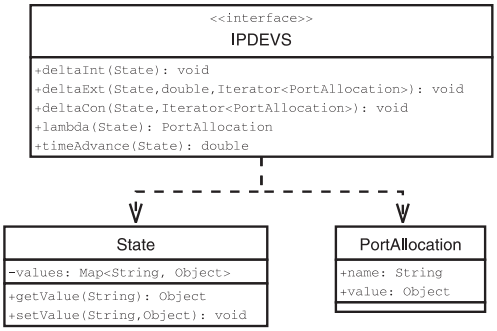

Figure 4: Description Elements Needed for Atomic PDEVS Models

We could now start to build a simulator for the bound models. However, such simulator would be far from being efficient due to the fact that most of the model descriptions are declarative.

4 EXECUTABLE MODELS

For being able to efficiently execute the specified models, declarative parts have to be transformed to an run-time optimized form. At this point the component-based design of the simulation layer within James II has proven beneficial.

4.1 The Simulation System James

James II (Himmelspach and Uhrmacher 2004a) is a modeling and simulation framework designed for being applicable to a wide range of applications. It has been applied in various domains such as social sciences (Ewert, Röhl, and Uhrmacher 2003), microbiology (Degenring, Röhl, and Uhrmacher 2004), and Artificial Intelligence (Schattenberg and Uhrmacher 2001).

The high flexibility of James II is achieved by consequently applying a modular design policy. Modeling and simulation layer are strictly separated. Starting from a parallel discrete event-oriented system specification, specific functionality such as support for dynamic structures or integration of external processes can be added to a model (Himmelspach and Uhrmacher 2004b).

4.2 Transformation of Bound to Executable Models

For transforming the bound models to models executable by James II, we have once again to follow the type hierarchy provided by the original schema definition. Figure 5 shows the transformation components needed for bound PDEVS models.

Using programming languages that provide higher order functions, like reflection in the Java programming language, eases the implementation of the transformation, e.g. reflection is used for instantiating the state variables of atomic models.

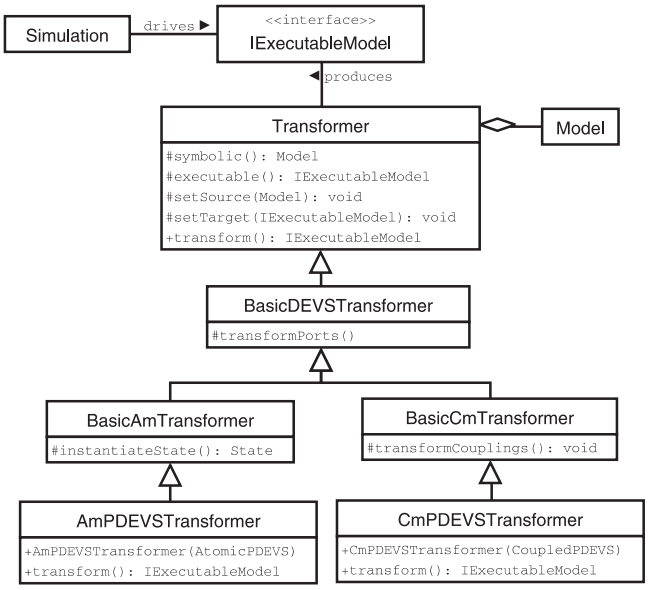

Figure 5: Transformation of Bound Models to Executable Ones

5 CONCLUSION

Our approach suggest to start integration of XML handling with a schema definition of a formalism, preferably a standard. Based on data binding XML's features can be exploited consequently for modeling and simulation systems by generating classes that can be integrated into the target simulation system. Checks at the editing time ensure part of the validity of constructed models. If the schema supports every kind of constraint checking needed for the formalisms the bare generated classes will do. The approach is open and allows to check for further constraints that are beyond XML's capabilities. As we demonstrated at the example of editors for generated PDEVS model classes, extensions can be added with very low manual coding effort.

However, generated model classes based on schema definitions still contain many declarative aspects that hamper an efficient execution, therefore a transformation into a runtime optimized representation is recommended. At this point we can exploit the mechanisms of James II with its different efficient simulators.

Generally, XML data binding symbiotically brings together declarative notations with imperative, tool specific implementations. Standard modeling formats can easily be integrated into simulation systems and changes in the format specification can be accounted for very rapidly on the tool side. Combining data binding with transformations forms the basis for the efficient execution of simulation models.

The approach has been implemented in James II and utilizes the component-based simulation engine for execution. However, it is applicable to simulation systems in general and will prove particularly beneficial in communities that work

with standard exchange formats or are interested in developing standards.

ACKNOWLEDGMENTS

This research is supported by the DFG (German Research Foundation).

REFERENCES

Billington, J., S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. 2003, June. The petri net markup language: Concepts, technology, and tools. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003), Eindhoven, The Netherlands, June 23-27, 2003 — Volume 2679 of LNCS / Wil M. P. van der Aalst and Eike Best (Eds.)*, 483–505: Springer-Verlag.

Bourret, R. 2005, April. XML data binding resources. Available online via <www.rpbourret.com/xml/XMLDataBinding.htm> [accessed July 11, 2005].

Brodkin, S. 2001, December. Use XML data binding to do your laundry: Explore JAXB and Castor from the ground up. Available online via <www.javaworld.com/javaworld/jw-12-2001/jw-1228-jaxb.html> [accessed July 11, 2005].

Degenring, D., M. Röhl, and A. M. Uhrmacher. 2004. Discrete event, multi-level simulation of metabolite channeling. *BioSystems* 75 (1-3): 29–41.

Ewert, U. C., M. Röhl, and A. M. Uhrmacher. 2003. *Agent based computational demography*, Chapter What good are deliberative interventions in large scale disasters? Exploring the consequences of crisis managment in premodern towns with agent-oriented simulation. Physica Verlag (Springer).

Filippi, J.-B., and P. Bisgambiglia. 2002. Enabling large scale and high definition simulation of natural systems with vector models and JDEVS. In *Proceedings of the 2002 Winter Simulation Conference*, 1964–1970.

Fishwick, P. A. 2002. Using XML for simulation modeling. In *Proceedings of the 2002 Winter Simulation Conference*, 616–622.

Harold, E. R. 2002. *Processing XML with Java*. Pearson Education.

Himmelspach, J., and A. M. Uhrmacher. 2004a. A component-based simulation layer for JAMES. In *Proc. of the 18th Workshop on Parallel and Distributed Simulation (PADS), May 16-19, 2004, Kufstein, Austria*, 115–122.

Himmelspach, J., and A. M. Uhrmacher. 2004b, October. Processing dynamic PDEVS models. In *Proceedings of the 12th IEEE International Symposium on MASCOTS*, ed. D. DeGroot and P. Harrison, 329–336. Volendam, The Netherlands: IEEE Computer Society.

Hong, K. J., T. G. Kim, and I. S. Kwon. 2000. DEVSIF: a relational algebraic DEVS intermediate format. In *AIS'2000*. Tucson, Arizona.

Java.net 2005. JAXB RI 2.0 early access. Available online via <https://jaxb.dev.java.net/jaxb20-ea> [accessed July 11, 2005].

McLaughlin, B. 2000, December. Validation with Java and XML Schema, part 4. *JavaWorld*. Available online via <www.javaworld.com/javaworld/jw-12-2000/jw-1208-validation4.html> [accessed July 11, 2005].

McLaughlin, B. 2002, May. Why data binding matters. Available online via <www.onjava.com/pub/a/onjava/2002/05/15/databind.html> [accessed July 11, 2005].

MoBIES 2004, February. Hybrid Systems Interchange Format (HSIF). Available online via <www.isis.vanderbilt.edu/Projects/mobies/download.asp> [accessed July 11, 2005].

Ort, E., and B. Mehta. 2003, March. Java Architecture for XML Binding (JAXB). Available online via <java.sun.com/developer/technicalArticles/WebServices/jaxb> [accessed July 11, 2005].

Schäfer, A. 2004, March. Visualisierung und XML-Darstellung von DEVS-Modellen. Master's thesis, Universität der Bundeswehr München.

Schattenberg, B., and A. M. Uhrmacher. 2001, February. Planning agents in James. *Proceedings of the IEEE* 89 (2): 158–173.

SISO 2005, March. Simulation interoperability standards organization's (SISO). Available online via <www.sisostds.org> [accessed July 11, 2005].

Sun 2005. Java architecture for xml binding (JAXB). Available online via <java.sun.com/xml/jaxb> [accessed July 11, 2005].

Tolk, A. 2004, December. XML mediation services utilizing model based data management. In *SCS Winter Simulation Conference*, ed. R. Ingalls, M. Rossetti, J. Smith, and B. Peters. Arlington, VA.

W3C 1999, November. XSL transformations (XSLT) version 1.0. Available online via <www.w3.org/TR/xslt> [accessed July 11, 2005].

Wang, Y.-H., and Y.-C. Lu. 2002. An XML-based DEVS modeling tool to enhance simulation interoperability. In *ESS 2002*.

Ware, T. K. 2005, March. LMX: W3C XML Schema to C/C++ data binding tool. Available online via <www.tech-know-ware.com/lmx> [accessed July 11, 2005].

AUTHOR BIOGRAPHIES

MATHIAS RÖHL holds a MSc in Computer Science from the University of Rostock. His research interests are on component-based modeling and agent-oriented simulation. He is currently a research scientist at the Modeling and Simulation Group at the University of Rostock. His e-mail address is <mroehl@informatik.uni-rostock.de> and his Web address is <http://www.informatik.uni-rostock.de/~mroehl>.

ADELINDE M. UHRMACHER is an Associate Professor at the Department of Computer Science at the University of Rostock and head of the Modeling and Simulation Group. Her research interests are in modeling and simulation methodologies, particularly agent-oriented modeling and simulation and their applications. Her e-mail address is <lin@informatik.uni-rostock.de> and her Web page is <http://www.informatik.uni-rostock.de/~lin>.