Representing Product Family Architectures in an Extensible Architecture Description Language

Eric M. Dashofy and André van der Hoek

Institute for Software Research
University of California, Irvine
Irvine, CA 92612–3425 USA

{edashofy,andre}@ics.uci.edu

Abstract. Product family architectures need to be captured much like "regular" software architectures. Unfortunately, representations for product family architectures are scarce and a deep understanding of all of the necessary features of such representations is still lacking. In this paper, we introduce an extensible XML-based representation that is suitable as a basis for rapidly defining new representations for product family architectures. We describe some of the details of this representation and present how Koala and Mae, two early representations for product family architectures, can be mapped onto our XML-based representation with relatively little effort.

1 Introduction

Product family architecture [1] research is following a path similar to that of software architecture research. Specifically, whereas early efforts have focused on understanding the role of product family architectures in the broader software development process and defining its dimensions of relevance, we are now in a stage where accurately capturing and representing product family architectures is becoming more important. Thus, we need a way to represent product family architectures. Because the canonical set of features of such a representation is unknown and because its identification requires a significant amount of research, we can expect a proliferation of product family architecture representations much like the proliferation of traditional software architecture description languages (ADLs).

The proliferation of architecture description languages carried with it a number of undesirable effects. While most languages contribute at least one unique feature, all share many concepts that are essentially the same [8]. Since each language (and its associated support tools) was created in a standalone fashion, significant duplication of effort resulted. This duplication not only occurred during the initial creation of each language; as new features were introduced in one language, other languages followed suit and provided similar functionality. An interchange language, Acme [6], was designed to partially remedy this situation, but its growth into an all-

encompassing mediating service never has taken place (in fact, Acme should probably be considered as a separate architecture description language altogether).

To avoid this kind of problem in the domain of product family architectures, we offer the following observations.

1. We can expect a significant amount of commonality among representations for product family architectures, as evidenced by the set of features shared by Koala [11] and Mae [10], two of the first such representations.
2. Representations for product family architectures can be viewed as representations for software architectures that are extended with some features designed specifically for capturing product family aspects of those architectures.
3. XML [2] schemas provide a useful platform upon which to build extensible, modular representations.

In this paper, we describe our efforts in building an extensible, XML-based representation for product family architectures, xADL 2.0 [4]. Key contributions of xADL 2.0 are its set of modular XML schemas that provide a basic framework for modeling product family architectures, its extensibility to allow future additions to (and modifications of) elements in the representation, and its associated tool support to automatically generate APIs for manipulating specific instances of product family architectures. xADL 2.0, thus, can serve as a basis for future research in product family architecture representations while avoiding the duplication of effort problem.

It is important to note that our work is not an attempt to unify representations for product family architectures. Rather, it provides a common, extensible representation onto which new modeling constructs can be efficiently added. This greatly reduces the duplication of effort that would result from building an entirely new product family architecture representation from scratch.

2 Existing Representations

Koala [11] and Mae [10] are two of the first representations for product family architectures. Both have adopted the philosophy that a product family architecture is considered a “normal” software architecture that contains several well-defined variation points. To capture these variation points, both have created special language features (Koala as additions to Darwin [7]; Mae as additions to an abstract language that encapsulates a mix of best practices in architecture description languages). Three specific differences exist between the two representations: (1) whereas Mae uses a new type of component (a variant component) to handle structural variations, Koala uses a design pattern (a switch); (2) whereas Mae strictly enforces identical sets of interfaces among variant components, Koala allows the set of interfaces to differ as long as a new variant subsumes the interfaces of the older one; and (3) whereas Mae tightly integrates version management in the representation, Koala relies on the use of an external configuration management system to capture the evolution of a product family architecture. As more representations for product family architectures are created, we can expect to see more differences, fueled by different needs and tradeoffs.

3 XML Schemas

XML schemas [5], recently ratified as a recommendation by the World Wide Web consortium (W3C), provide a rich and extensible meta-language with which to create new representation formats. The original meta-language provided by XML was the Document Type Definition (DTD). DTDs allow authors to specify what elements and attributes may be present in a conforming document. XML schemas are more expressive than DTDs, primarily because they provide a much richer type system that offers significantly better support for developing modularly-extensible languages. XML schema authors can define base types in one schema and extend those types in other schemas, adding, changing, and, in some cases, removing elements or attributes in the extended types. Unlike DTDs, this does not require modifications to the base schema.

4 Approach

Leveraging XML schemas, we developed xADL 2.0 specifically to be an extensible and modular representation for product family architectures. xADL 2.0 consists of a set of core XML schemas that capture the basic elements of a product family architecture, such as components, connectors, versions, variants, and options. Designers leverage these elements in creating their own representations, thereby avoiding duplication and minimizing their effort. This practice of modularity and extensibility is even reflected in the core set of xADL 2.0 schemas. The three key elements of a product family representation, namely versions, options, and variants, are provided as independent extensions to a set of schemas that define a "normal" software architecture. Product family architects can choose any combination of these features as needed.

In xADL 2.0, optional elements are perhaps the simplest to understand. Optionality is a property of a system's structure. An optional element (component, connector, or link) in the architecture may or may not get instantiated when the system is executed. Whether or not the element is instantiated depends on whether an associated guard condition is satisfied. In keeping with the modularly extensible nature of xADL 2.0, the definition of "optional" does not restrict the format of the guard condition. We have provided a Boolean guard extension to xADL 2.0 that allows designers to specify Boolean conditions in XML.

Variants in xADL 2.0 are represented as union types. Recall that in a programming language, a variable of a union type may take on one of many actual types when it is instantiated. Similarly, in xADL 2.0, a structural component or connector of a variant type may take on one of many actual types depending on the satisfaction of guard conditions. Each possible variant is associated with a guard. All guards for a set of variants must be mutually exclusive. When the system is executed, guards are evaluated and concrete types are chosen for any component or connector of a variant type.

In xADL 2.0, types are the versioned entities. It is possible, in xADL 2.0, to represent a full version graph for any component, connector, or interface type. These graphs are represented as a collection of linked nodes, placing little restriction on what nodes can be linked to what other nodes. This provides maximal flexibility when choosing a version graph representation, and the highest chance for successfully integrating such a representation with an existing, off-the-shelf CM system. Because we chose to version types, it is possible to have multiple versions of the same component or connector running in a single system. For instance, a system might contain a component of type T version 1.1 as well as a separate component of type T version 2.0. While technically types T version 1.1 and T version 2.0 are considered different types, it is useful to understand how they are related in terms of their evolution. Furthermore, designers using xADL 2.0 can elect to put more strict restrictions on how types may evolve. For instance, a designer might institute a policy that all later versions of a type must expose at least the same interface signatures as their predecessors, guaranteeing backward compatibility at the interface level. Such complex restrictions cannot be enforced solely by XML syntax; as such, external tools must enforce them. Furthermore, it is important to note that the versioning schema does not inherently restrict what kind of metadata can be included in the version graph. We fully expect designers working with xADL 2.0 to write small extensions to the versions schema that allow many different kinds of metadata to reside there.

As noted above, designers can select any number of these extensions and use them as they are needed. A designer working on the initial stages of a product family architecture may want to avoid including versioning information until the basic points of variation and optionality are worked out. In xADL 2.0, this is entirely possible—the designer would simply avoid using the versions schema until it was needed. On the other hand, advanced designers who want to encode a lot of information in their product line representation can create extensions to the xADL 2.0 core schemas, adding new modeling constructs that fit their particular domain of interest.

5 Two Examples

To illustrate how our representation can be used in creating specific product family architecture representations, we have taken Koala and Mae's unique representation characteristics and mapped them onto XML schemas that extend, in minor ways, our existing xADL 2.0 schemas. This shows how using xADL 2.0 as a baseline can significantly reduce duplication of effort. Below is a brief discussion of each mapping.

5.1 Koala

Four issues need to be resolved in mapping Koala onto xADL 2.0. The first is straightforward: since Koala relies on an external configuration management system

to provide version management for an evolving product family architecture, we exclude the versioning schema from the base set of schemas upon which we built the Koala representation. Because of the independence and modularity of the schemas, this is a trivial operation.

The second issue regards component variability. Each component in Koala that exhibits variability has an associated special kind of interface through which selections are performed to choose which variant appears in an actual system incarnation. These special kinds of interfaces are called *diversity* interfaces. To model these interfaces, an extra XML schema was written. The following captures the canonical functionality of this schema (some tags are omitted for clarity; full schemas are shown in Appendix A):

```
<xsd:schema xmlns="diversity.xsd">
 <xsd:complexType name="DiversityInterface">
  <xsd:complexContent>
   <xsd:restriction base="types:Interface">
    <xsd:sequence>
          ...
     <!--This is the only element that changes-->
     <xsd:element name="direction"
          type="archinstance:Direction"
          minOccurs="0" maxOccurs="1"
          fixed="out"/>
          ...
    </xsd:sequence>

    <xsd:attribute name="id"
         type="archinstance:Identifier"/>
   </xsd:restriction>
  </xsd:complexContent>
 </xsd:complexType>

 <xsd:complexType name="DiversityComponentType">
  <xsd:complexContent>
   <xsd:extension base="types:ComponentType">
    <xsd:sequence>
     <xsd:element name="diversity"
          type="DiversityInterface"
          minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:schema>
```

This schema first defines a diversity interface to be just like any other interface, except that it enforces the Koala rule that diversity interfaces always be required (“out”) interfaces. Then, the schema defines an extended version of a component

type to include a single diversity interface. Whenever a component with variability needs to be modeled, an XML element of this new component type should be used.

The third issue regards mapping switches onto the XML schemas. In Koala, switches are used to support a pattern of components in which one component can use one of two other components depending on the desired configuration as defined by a diversity interface. As such, a switch can be defined as a connector that connects three components and has one diversity interface to determine which of two components is used by the main component of the switch construction. Once again, we drafted an XML schema. Shown below, it is of note that this schema is of a similarly simple nature as the schema above—demonstrating the ease of extensibility in xADL 2.0.

```
<xsd:schema xmlns="switch.xsd">
 <xsd:complexType name="SwitchConnector">
  <xsd:complexContent>
   <xsd:extension base="types:Connector">
    <xsd:sequence>
     <xsd:element name="selector"
          type="diversity:DiversityInterface"
          minOccurs="1" maxOccurs="1"/>
     <xsd:element name="inPort"
         type="archInstance:Interface"
          minOccurs="1" maxOccurs="1"/>
     <xsd:element name="outPort"
          type=archInstance:Interface"
          minOccurs="2" maxOccurs="2"/>
    </xsd:sequence>
   </xsd:restriction>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:schema>
```

The final issue is that Koala does not have connectors; however, they are part of our most basic XML schemas. As demonstrated by the remedies to the previous two issues, our schemas are extensible and modular at the macro level (i.e. individual types in a schema). However, they also exhibit flexibility at the micro level: certain critical elements in the XML schemas may or may not be used depending on the particular need. In this case, simply not putting any connectors in an instance of the representation and establishing links among interfaces on components rather than among components and connectors solves the problem. The representation is general enough to allow this.

5.2 Mae

Since xADL 2.0 was partially inspired by the predecessor to Mae (Ménage [9]), the mapping of Mae onto xADL 2.0 is not as complicated as Koala's mapping. Two additional items need to be modeled: subtype relations and styles. Although one could write a single extension that adds both at once, each should be added in its

own extension—staying with the modular philosophy of xADL 2.0. To add style designations, the following simplified XML schema was written:

```
<xsd:schema xmlns="style.xsd">
 <xsd:complexType name="StyleComponentType">
  <xsd:complexContent>
   <xsd:extension base="types:ComponentType">
    <xsd:sequence>
      <xsd:element name="style" type="xsd:string"
           minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:schema>
```

Similarly, the following schema adds subtype relations by adding two fields to the definition of a component type: a link to the base type and a designation of the kind of subtype relation (see [10]).

```
<xsd:schema xmlns="subtype.xsd">
 <xsd:complexType name="SubtypeComponentType">
  <xsd:complexContent>
   <xsd:extension base="types:ComponentType">
    <xsd:sequence>
     <xsd:element name="baseType"
          type="archInstance:XMLLink"
          minOccurs="0" maxOccurs="1"/>
     <xsd:element name="subTypeRelation"
          type="xsd:string"
          minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:schema>
```

Interoperability is a concern given these two schemas. Since XML does not provide multiple inheritance, these two schemas can only co-exist if the sets of component types covered by the extensions are mutually exclusive. This means that a component type cannot have both a subtype and style, weakening our extensibility through modularity argument. We, thus, have introduced artificial dependencies as a solution (see [4]). For instance, in the above case, we could make the subtype extension an extension of the style extension, or vice-versa. Because both the style and subtype tags are optional, it is possible to have a component type with a subtype, a style, or both. Nonetheless, this is not an ideal solution and we await the advent of multiple inheritance in the XML standard—something that is being discussed in the XML schema W3C committee.

6 Tool Support

Extensibility of the XML schemas, by itself, is not sufficient to provide an adequate solution to our problem; we need associated tool support to make it easy for developers of product family representations to leverage and extend the schemas without having to implement advanced XML capabilities. Moreover, it is important that the tools provide access to the schemas in the same modular and extensible fashion that underlies the schemas themselves.

We have developed a code generator that takes a set of xADL 2.0 XML schemas as input and generates a Java library for each of the schemas as output. The libraries hide all XML details from their users and provide access to the elements in the schema in a natural fashion (e.g., methods such as `addInterface` and `setDescription` that directly reflect the entities in the schemas). Additionally, the libraries that are generated interoperate: they provide an object hierarchy that maps one-to-one to the hierarchy of elements in the XML schemas. Furthermore, the libraries' ability to ignore unknown XML tags allows seamless interoperation of applications that use the libraries to manage a xADL 2.0 document. An application that manipulates component types does not need to be aware of the fact that the component type actually has subtype and style elements used by some other application. Further details on these and other important considerations are provided in another paper [3].

7 Conclusions

Research in the area of product family architecture is still maturing at a rapid pace. It would be imprudent at this early stage in the development of product family architecture representations to attempt to define a universal or all-encompassing representation format. Simply put, we do not know all the aspects of a product family that we may want to capture. Yet, we want to leverage early representations as much as possible in our future explorations. In this paper we have presented aspects of xADL 2.0, a modular and extensible representation for product family architectures that makes this possible. Leveraging XML schemas and a flexible code generator, xADL 2.0 provides developers with the ability to quickly and easily create new representations that explore new directions in product family architecture research. We have demonstrated this ability through our mappings of two existing representations for product family architectures. Our future work includes the development of additional XML schemas to provide an increasing set of "standard" product family architecture features, experimentation with new features for product family architectures, and further dissemination of the schemas and toolset to allow an increasing set of other parties to develop XML schemas for xADL 2.0.

8 URL

More information about xADL 2.0 can be found here:

http://www.isr.uci.edu/projects/xarchuci/

9 Acknowledgements

The authors would like to thank Richard N. Taylor for supporting the authors in this work.

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

10 References

[1] Batory, D., *Product-Line Architectures*, in *Invited Presentation: Smalltalk and Java in Industry and Practical Training*. 1998.

[2] Bray, T., et al., *Extensible Markup Language (XML)*.

[3] Dashofy, E.M., *Issues in Generating Data Bindings for an XML Schema-Based Language*, in *Proceedings of XML Technologies and Software Engineering*. 2001.

[4] Dashofy, E.M., van der Hoek, A., and Taylor, R.N., *A Highly-Extensible, XML-Based Architecture Description Language*, in *Working IEEE/IFIP Conference on Software Architecture*. 2001 (to appear).

[5] Fallside, D.C., *XML Schema Part 0: Primer*. 2000.

[6] Garlan, D., Monroe, R.T., and Wile, D., *ACME: An Architectural Interconnection Language*. 1995, Carnegie Mellon University: Pittsburgh, PA.

[7] Magee, J., et al., *Specifying Distributed Software Architectures*, in *Proceedings of the Fifth European Software Engineering Conference*. 1995.

[8] Medvidovic, N. and Taylor, R.N., *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, 2000. **26**(1): p. 70-93.

[9] van der Hoek, A., *Capturing Product Line Architectures*, in *Proceedings of the Fourth International Software Architecture Workshop*. 2000. p. 95-99.

[10] van der Hoek, A., et al., *Taming Architectural Evolution*, in *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. 2001 (to appear).
[11] van Ommering, R., et al., *The Koala Component Model for Product Families in Consumer Electronics Software*. IEEE Computer, 2000. **33**(2): p. 78-85.

Appendix A

This appendix lists the XML Schemas defined above in full, with no omitted tags or declarations.

A.1 diversity.xsd

```
<xsd:schema xmlns="http://www.ics.uci.edu/pub/arch/xArch/diversity.xsd"
        xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:archinstance="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
        xmlns:types="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
        targetNamespace="http://www.ics.uci.edu/pub/arch/xArch/diversity.xsd"
        elementFormDefault="qualified"
        attributeFormDefault="qualified">

 <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
    schemaLocation="http://www.ics.uci.edu/pub/arch/xarch/schemas/instance.xsd"/>
 <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
    schemaLocation="http://www.isr.uci.edu/projects/xarchuci/ext/types.xsd"/>

 <xsd:complexType name="DiversityInterface">
  <xsd:complexContent>
   <xsd:restriction base="types:Interface">
    <xsd:sequence>
     <xsd:element name="description"
        type="archinstance:Description"/>
     <!-- This is the only element that changes -->
     <xsd:element name="direction"
        type="archinstance:Direction"
        minOccurs="0" maxOccurs="1"
        fixed="out"/>
     <xsd:element name="type"
        type="archinstance:XMLLink"
        minOccurs="0" maxOccurs="1"/>
```

```
      </xsd:sequence>
      <xsd:attribute name="id" type="archinstance:Identifier"/>
    </xsd:restriction>
  </xsd:complexContent>
 </xsd:complexType>

 <xsd:complexType name="DiversityComponentType">
  <xsd:complexContent>
   <xsd:extension base="types:ComponentType">
    <xsd:sequence>
     <xsd:element name="diversity"
         type="DiversityInterface"
         minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:schema>
```

A.2 switch.xsd

```
<xsd:schema xmlns="http://www.ics.uci.edu/pub/arch/xArch/switch.xsd"
          xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
          xmlns:xlink="http://www.w3.org/1999/xlink"
          xmlns:archinstance="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
          xmlns:types="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
          xmlns:diversity="http://www.ics.uci.edu/pub/arch/xArch/diversity.xsd"
          targetNamespace="http://www.ics.uci.edu/pub/arch/xArch/switch.xsd"
          elementFormDefault="qualified"
          attributeFormDefault="qualified">

 <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
     schemaLocation="http://www.ics.uci.edu/pub/arch/xarch/schemas/instance.xsd"/>
 <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
     schemaLocation="http://www.isr.uci.edu/projects/xarchuci/ext/types.xsd"/>
 <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/diversity.xsd"
     schemaLocation="http://www.isr.uci.edu/projects/xarchuci/ext/diversity.xsd"/>

<xsd:complexType name="SwitchConnector">
 <xsd:complexContent>
  <xsd:extension base="types:Connector">
   <xsd:sequence>
    <xsd:element name="selector"
        type="diversity:DiversityInterface"
```

```
        minOccurs="1" maxOccurs="1"/>
    <xsd:element name="inPort"
        type="archInstance:Interface"
        minOccurs="1" maxOccurs="1"/>
    <xsd:element name="outPort"
        type=archInstance:Interface"
        minOccurs="2" maxOccurs="2"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

</xsd:schema>
```

A.3 style.xsd

```
<xsd:schema xmlns="http://www.ics.uci.edu/pub/arch/xArch/style.xsd"
          xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
          xmlns:xlink="http://www.w3.org/1999/xlink"
          xmlns:archinstance="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
          xmlns:types="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
          targetNamespace="http://www.ics.uci.edu/pub/arch/xArch/style.xsd"
          elementFormDefault="qualified"
          attributeFormDefault="qualified">

  <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
      schemaLocation="http://www.ics.uci.edu/pub/arch/xarch/schemas/instance.xsd"/>
  <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
      schemaLocation="http://www.isr.uci.edu/projects/xarchuci/ext/types.xsd"/>

  <xsd:complexType name="StyleComponentType">
    <xsd:complexContent>
      <xsd:extension base="types:ComponentType">
        <xsd:sequence>
          <xsd:element name="style" type="xsd:string"
              minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

</xsd:schema>
```

A.4 subtype.xsd

```
<xsd:schema xmlns="http://www.ics.uci.edu/pub/arch/xArch/subtype.xsd"
          xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
          xmlns:xlink="http://www.w3.org/1999/xlink"
          xmlns:archinstance="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
          xmlns:types="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
          targetNamespace="http://www.ics.uci.edu/pub/arch/xArch/subtype.xsd"
          elementFormDefault="qualified"
          attributeFormDefault="qualified">

  <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
      schemaLocation="http://www.ics.uci.edu/pub/arch/xarch/schemas/instance.xsd"/>
  <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
      schemaLocation="http://www.isr.uci.edu/projects/xarchuci/ext/types.xsd"/>

  <xsd:complexType name="SubtypeComponentType">
   <xsd:complexContent>
    <xsd:extension base="types:ComponentType">
     <xsd:sequence>
      <xsd:element name="baseType"
          type="archinstance:XMLLink"
          minOccurs="0" maxOccurs="1"/>
      <xsd:element name="subTypeRelation"
          type="xsd:string"
          minOccurs="0" maxOccurs="1"/>
     </xsd:sequence>
    </xsd:extension>
   </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```