Process Representation Using Architectural Forms: Accentuating the Positive

The PSL (Process Specification Language) project is creating a standard language for process specification to serve as an interlingua to integrate multiple process−related applications throughout the manufacturing life cycle. This interchange language is unique due to the formal semantic definitions (the ontology) that underlie the language. The PSL ontology is organized modularly with a small set of core concepts and multiple extensions which add to the core. We are developing a mapping from the PSL semantic concepts to XML (Extensible Markup Language) that uses architectural forms to explicitly specify the relationship between the concepts in a process specification and the PSL core and extensions. An example showing architectural form processing of an XML−encoded process specification demonstrates the usefulness of architectural forms for managing modular specifications, mapping non−PSL syntax to PSL terminology, and generating extension−specific data views.

Joshua Lubell, National Institute of Standards and Technology
Craig Schlenoff, National Institute of Standards and Technology

Introduction

Ac-cent-tchu-ate the positive, e-lim-my-nate the negative,
Latch on to the affirmative, don't mess with Mister In-between.

This refrain, from the 1940s hit song *Ac−Cent−Tchu−Ate the Positive* [MER 44], nicely summarizes the benefits of architectural forms [ISO 10744] [ISO 10744−1] [MEG 98]. Architectural forms enable XML (Extensible Markup Language) [W3C 98] or SGML (Standard Generalized Markup Language) [ISO 8879] [GOL 92] applications to easily create architecture−specific document views, retaining only relevant markup and character data while hiding all other document content. Architectural forms also support the substitution of identifier names in a base architecture for names in the client document, facilitating data sharing between user communities with inconsistent terminologies. Such capabilities are possible because architectural forms unambiguously specify how document types relate to one another.

This paper discusses how we are using architectural forms in an XML representation of the proposed PSL (Process Specification Language) [SCH 99−2] [SCH 99−1] [PSL] standard. We begin with an overview of PSL and how it can be presented as XML. Next we provide an example showing our use of architectural forms. Then we summarize the benefits of our approach and discuss some related issues. We assume the reader is already familiar with architectural forms and has seen examples of them being used. If this is not the case, the reader would be well advised to read some introductory material such as chapters 9 and 10 of David Megginson's book *Structuring XML Documents* [MEG 98] or Eliot Kimber's on−line tutorial [KIM 97] before proceeding further. We do not assume prior knowledge of PSL.

Overview of PSL

Many applications use process information, including production scheduling, process planning, workflow, business process re–engineering, simulation, process realization process modeling, and project management. The problem is that all of these applications represent process information in their own internal representations. Therefore communication between them, a growing need for industry, is nearly impossible without some kind of translator. The PSL project at NIST (National Institute of Standards and Technology) is working with industry and academia to create a standard language for process specification in order to integrate multiple process–related applications throughout the manufacturing life cycle. This interchange language is unique due to the formal definitions (the ontology) that underlie the language. Because of these explicit and unambiguous definitions, information exchange can be achieved without relying on hidden assumptions or subjective mappings. PSL semantics are represented using KIF (Knowledge Interchange Format) [GEN 92]. Briefly stated, KIF is a formal language developed for the exchange of knowledge among disparate computer programs. KIF provides the level of rigor necessary to unambiguously define concepts, a necessary characteristic to exchange manufacturing process information using the PSL ontology.

The primary components of PSL are definitions and relations expressed in KIF. These definitions and relations constitute the PSL ontology for processes. We can include an infinite set of terms in our ontology, but they can only be shared if we agree on their definitions. It is the definitions that are being shared, not simply the terms.

The PSL ontology consists of definitions and axioms for the concepts of PSL. However, this is not simply an amorphous set of sentences. Figure 1 gives an overview of the semantic architecture of the PSL ontology. There are three major components —the axioms of PSL Core, core theories, and definitions that are organized as sets of extensions to PSL Core. The PSL ontology is organized modularly in order to facilitate the addition of new extensions as future industrial requirements for PSL emerge. PSL's modularity also makes it possible for applications to support a subset of extensions responding to a particular class of process specifications without having to support the entire PSL ontology.

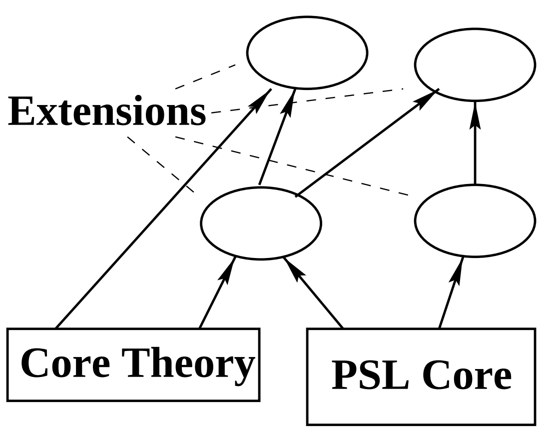

Figure 1. The PSL Semantic Architecture

PSL Core specifies the semantic primitives in the PSL ontology corresponding to the fundamental intuitions about activities. Primitives are those terms for which we do not give definitions; rather, we specify axioms that constrain the interpretation of the terms. An example of such a primitive is the "before" relation. An example of an axiom constraining the interpretation of "before" is the statement that the "before" relation is irreflexive. This axiom is formally specified in KIF as follows:

```
(forall (?p)
        (not (before ?p ?p)))
```

The terms that have definitions can be grouped into extensions of PSL Core. The extensions are organized by logical dependencies —one extension depends on another if the definitions of any terms in the first extension require terms defined in the second extension. PSL Core is therefore intended to be used as the basis for defining terms of the extensions in the PSL ontology. PSL extensions often constrain terms in the PSL Core to define new terms. In addition to PSL Core and its extensions, other sets of axioms may be required which introduce new primitive concepts; these axioms are grouped into core theories. Although not extensions themselves, these core theories provide building blocks necessary to define concepts in PSL. The extensions that introduce new primitive concepts do so because the concepts introduced in the PSL Core are not sufficient for defining the terms introduced in the extension. Therefore, new primitive concepts are introduced within the extensions to ensure that all other terms within the extension can be completely defined.

PSL and XML

We are developing a mapping from the PSL semantic concepts to XML. The most obvious reason for this endeavor is XML's popularity. Vendors of mainstream software applications such as Internet browsers, database environments, and business productivity tools are either already supporting or intend to support XML in their products. Mapping PSL instances to XML will enable process specifications to be interpreted by these generic applications, lowering the barriers to data sharing. Another reason to map PSL to XML is XML's "tag−centric" syntax. XML is a natural fit for representing ordered sequences and hierarchies. Thus it is well suited for describing PSL's ordering and sub−activity relationships.

Although XML's ubiquity and tag−oriented syntax make it a useful presentation format for PSL, XML has one major weakness. XML is not as rich a representation as KIF. In particular, it can be difficult to describe arbitrary constraints between data elements in an information model in such a way that an XML application could enforce the constraints. For example, consider a manufacturing process involving replacement of an old, worn−out part with a newer part. The newer part does not necessarily have to be brand new, but its level of wear−and−tear must be some measurable quantity less than that of the old part in order to make the replacement process economically worthwhile.

Such a constraint could be represented as a relation in KIF, but there is no obvious way it can be represented in XML. Because XML is not as powerful as KIF when it comes to specifying constraints, XML's presentational abilities for PSL are limited. Exactly what those limitations are is a topic for future research, but intuitively it seems that some aspects of the PSL ontology specified as KIF relations must either be implicitly represented (as can be done with ordering and sub–activity relationships) or omitted in an XML presentation. Also, since portions of the ontology are hard to specify in XML, XML is not always suitable as an authoring environment for PSL. Still, despite XML's representational shortcomings, its advantages outweigh its disadvantages —particularly for applications not requiring the full power of the PSL ontology. Furthermore, efforts are underway to develop a schema language superior to the existing DTD (Document Type Definition) mechanism for defining the structure, content and semantics of XML documents [W3C 99–1] [W3C 99–2]. Once it becomes standardized, the XML Schema language will narrow the gap between XML and KIF in their respective abilities to represent constraints.

Use of Architectural Forms

PSL is modular by design, and any presentation of PSL should preserve this modularity. Thus our approach for representing PSL as XML involves specifying a DTD for PSL Core and DTDs for each PSL extension. An XML–encoded process specification then employs architectural forms to explicitly specify the PSL subset the process specification uses as well as how the process specification conforms to each DTD in the subset. As an example of this approach, we specify architectures for PSL Core and a PSL extension and then derive a client document from these two architectures.

PSL Core ontology contains four primitive classes, three primitive relations, and two primitive functions. The classes are object, activity, activity–occurrence and timepoint; the relations are participates–in, before, and occurrence–of; and the functions are beginof, and endof. Timepoints are assumed to be ordered by the before relation. We use the following XML DTD to represent this core ontology:

```
<!--                    PSL Core (pslcore.dtd)                 -->
<!ELEMENT  psl          (objects?, timepoints, activities,
                             occurrences)                       >
<!ELEMENT  objects      (object+)                               >
<!ELEMENT  object       (name, participations?)                 >
<!ATTLIST  object
             id         ID                            #REQUIRED >
<!ELEMENT  name         (#PCDATA)                               >
<!--                 occurrences in which this object participates -->
<!ELEMENT  participations
                        (participatesin+)                       >
<!ELEMENT  participatesin EMPTY                                 >
<!ATTLIST  participatesin
             occurrence IDREF                         #REQUIRED
             beginof    IDREF                         #REQUIRED
             endof      IDREF                         #REQUIRED >
<!ELEMENT  timepoints   (timepoint+)                            >
<!ELEMENT  timepoint    (#PCDATA)                               >
<!ATTLIST  timepoint
             id         ID                            #REQUIRED >
<!ELEMENT  activities   (activity+)                             >
<!ELEMENT  activity     (#PCDATA)                               >
<!ATTLIST  activity
             id         ID                            #REQUIRED >
<!ELEMENT  occurrences  (occurrence+)                           >
```

```
<!ELEMENT  occurrence   (#PCDATA)                                  >
<!ATTLIST  occurrence
             activity   IDREF                             #REQUIRED
             id         ID                                #REQUIRED >
```

The `object`, `timepoint`, `activity`, and `occurrence` elements represent the four corresponding primitive classes in the PSL Core ontology. `participatesin` represents the participates−in relation. The `activity` attribute of element `occurrence` corresponds to the occurrence−of relation. Attributes of element `participatesin` represent the functions beginof and endof. The ordering of `timepoint` elements in the content mode for `timepoints` implicitly represents the before relation. The character data inside `timepoint`, `activity`, and `occurrence` elements is intended to be interpreted by an application as documentation describing the corresponding PSL timepoint, activity, or activity−occurrence.

We now consider PSL's Activity Occurrences extension, whose purpose is to describe how activity−occurrences relate to one another with respect to the time at which they start and end. The following DTD can be used to represent this extension:

```
<!--                 Activity occurrences extension (actoccur.dtd) -->
<!ELEMENT  occur        (timepoints, occurrences)                  >
<!ELEMENT  timepoints   (timepoint+)                               >
<!ELEMENT  timepoint    (#PCDATA)                                  >
<!ATTLIST  timepoint
             id         ID                                #REQUIRED >
<!ELEMENT  occurrences  (occurrence+)                              >
<!ELEMENT  occurrence   (#PCDATA)                                  >
<!ATTLIST  occurrence
             beginof    IDREF                             #REQUIRED
             endof      IDREF                             #REQUIRED >
```

Like all PSL extensions, Activity Occurrences requires PSL Core axioms. Indeed the DTD uses the core classes timepoint and occurrence and the core functions beginof and endof. The elements corresponding to these core classes could have been derived using architectural forms from their counterparts in the PSL Core DTD. In order to keep this paper brief, however, we do not specify architectural forms as part of the Activity Occurrences DTD.

Now we consider a very simple XML−encoded process specification for building an automobile engine. Our choice of scenario is adapted from the larger and far more complicated Camile Motor Works manufacturing process interoperability scenario [POL 98] developed for PSL. Building consists of two steps: assembly followed by testing. To make this example more interesting, our process specification uses some non−PSL terminology. In particular it uses the following jargon from the IDEF3 Process Description Capture Method [IDEF3] [IDEF], a mechanism for modeling system behavior:

- UoB (Unit of Behavior) —analogous to an activity in PSL.
- Component —analogous to a sub−activity in PSL. Sub−activities are described in a PSL extension not included in our example.
- Process —analogous to an activity−occurrence in PSL.

We now supply the following architecture use declarations in order to make the PSL Core and Activity Occurrences DTDs into enabling architectures for our IDEF3−influenced process specification:

```
<?IS10744:arch name="psl"
        dtd-system-id="pslcore.dtd"
        suppressor-att="psl-processing"
        renamer-att="psl-atts"?>

<?IS10744:arch name="occur"
        dtd-system-id="actoccur.dtd"?>
```

These declarations declare two architectures, “psl” for PSL Core and “occur” for the Activity Occurrences extension. Neither declaration specifies a form attribute, so the name of the element attribute used for deriving elements defaults to the architecture name (“psl” for the PSL Core architecture and “occur” for the Activity Occurrences architecture). The psl architecture use declaration specifies additional element attributes for deriving attributes and selectively suppressing architectural form processing.

The following IDEF3–inspired DTD for our process specification is derived from these two architectures. Its architectural form attributes have `#FIXED` default values so that values do not have to be specified in the document instance. That way, architectural forms and PSL–specific terminology foreign to IDEF3 are hidden from document authors. The `psl` and `occur` attributes make explicit the correspondence between UoBs and processes in IDEF3 and activities and activity–occurrences in PSL. The `psl-atts` attributes specify that the process in which an IDEF3 object participates is semantically equivalent to the occurrence in which a PSL object participates and that the UoB associated with an IDEF3 process corresponds to the activity associated with a PSL occurrence. The value `SArcForm` assigned to attribute `psl-processing` of element `uob` suppresses PSL Core architectural processing for `uob`’s descendants. This is needed because sub–activities are outside the scope of the core ontology. Since character data is not suppressed, `SArcForm` has the effect of mapping the content inside the IDEF3 DTD’s `documentation` element to the content inside the PSL Core DTD’s `activity` element.

```
<!--     process specification using IDEF3 terminology (idef3.dtd) -->
<!ELEMENT  idef3          (objects?, timepoints, uobs, processes)     >
<!ELEMENT  objects       (object+)                                    >
<!ELEMENT  object        (name, participations?)                      >
<!ATTLIST  object
             id          ID                                 #REQUIRED >
<!ELEMENT  name          (#PCDATA)                                    >
<!ELEMENT  participations (participatesin+)                           >
<!ELEMENT  participatesin EMPTY                                       >
<!ATTLIST  participatesin
             process     IDREF                              #REQUIRED
             beginof     IDREF                              #REQUIRED
             endof       IDREF                              #REQUIRED
             psl-atts    CDATA             #FIXED "occurrence process" >
<!ELEMENT  timepoints    (timepoint+)                                 >
<!ELEMENT  timepoint     (#PCDATA)                                    >
<!ATTLIST  timepoint
             id          ID                                 #REQUIRED >
<!--                     Units of Behavior                            >
<!ELEMENT  uobs          (uob+)                                       >
```

```
<!ATTLIST  uobs
             psl       NMTOKEN                  #FIXED "activities"  >
<!ELEMENT  uob          (documentation, components?)                 >
<!ATTLIST  uob
             id         ID                                 #REQUIRED
             psl        NMTOKEN                  #FIXED "activity"
             psl-processing NMTOKEN              #FIXED "sArcForm" >
<!ELEMENT  documentation (#PCDATA)                                   >
<!ELEMENT  components   (component+)                                 >
<!ELEMENT  component    EMPTY                                        >
<!ATTLIST  component
             uob        IDREF                              #REQUIRED >
<!ELEMENT  processes    (process+)                                   >
<!ATTLIST  processes
             psl        NMTOKEN               #FIXED "occurrences"
             occur      NMTOKEN               #FIXED "occurrences" >
<!ELEMENT  process      (#PCDATA)                                    >
<!ATTLIST  process
             beginof    IDREF                              #REQUIRED
             endof      IDREF                              #REQUIRED
             uob        IDREF                              #REQUIRED
             id         ID                                 #REQUIRED
             psl        NMTOKEN                #FIXED "occurrence"
             occur      NMTOKEN                #FIXED "occurrence"
             psl-atts   CDATA                #FIXED "activity uob" >
```

The following client document conforms to this DTD. Architectural processing attributes are hidden from view, thanks to the use of `#FIXED` attributes in the DTD. The client document specifies an automobile engine object and a set of three ordered timepoints specifying the beginning state, the completion of assembly, and the completion of testing. There is a UoB for building an artifact with constituent UoBs for assembly and testing. Each of these UoBs occurs as a process making use of the automobile engine object.

```
<?xml version="1.0"?>
<!DOCTYPE idef3 SYSTEM "idef3.dtd">
<idef3>
  <objects>
    <object id="o1">
      <name>automobile engine</name>
      <participations>
        <participatesin process="p1" beginof="t1" endof="t3"/>
        <participatesin process="p2" beginof="t1" endof="t2"/>
        <participatesin process="p3" beginof="t2" endof="t3"/>
      </participations>
    </object>
  </objects>
  <timepoints>
    <timepoint id="t1">start</timepoint>
    <timepoint id="t2">assembly complete</timepoint>
    <timepoint id="t3">everything complete</timepoint>
  </timepoints>
  <uobs>
    <uob id="u1">
      <documentation>build</documentation>
```

```
      <components>
        <component uob="u2"/>
        <component uob="u3"/>
      </components>
    </uob>
    <uob id="u2">
      <documentation>assemble</documentation>
    </uob>
    <uob id="u3">
      <documentation>test</documentation>
    </uob>
  </uobs>
  <processes>
    <process beginof="t1" endof="t3" uob="u1" id="p1">build automobile
engine</process>
    <process beginof="t1" endof="t2" uob="u2" id="p2">assemble engine</process>
    <process beginof="t2" endof="t3" uob="u3" id="p3">test engine</process>
  </processes>
</idef3>
```

Now that all of the necessary ingredients are in place —the PSL Core and Activity Occurrences meta–DTDs, their corresponding architecture use declarations, and an IDEF3–inspired client document conforming to its own DTD as well as the base architectures defined using the meta–DTDs —an architecture engine can generate architectural documents conforming to the PSL Core and Activity Occurrences DTDs. The PSL Core architectural document generated by the architecture engine is as follows. Note that UoBs have become activities, processes have become occurrences, and the concept of components has been suppressed. The `beginof` and `endof` attributes of `process` are suppressed as well, since this information is part of the Activity Occurrences extension and is outside the scope of PSL Core. Objects, timepoints, and the notion of participation are preserved since these concepts appear in both the IDEF3 and PSL Core DTDs.

```
<?xml version="1.0"?>
<!DOCTYPE psl SYSTEM "pslcore.dtd">
<psl>
  <objects>
    <object id="O1">
      <name>automobile engine</name>
      <participations>
        <participatesin occurrence="p1" beginof="t1" endof="t3"/>
        <participatesin occurrence="p2" beginof="t1" endof="t2"/>
        <participatesin occurrence="p3" beginof="t2" endof="t3"/>
      </participations>
    </object>
  </objects>
  <timepoints>
    <timepoint id="t1">start</timepoint>
    <timepoint id="t2">assembly complete</timepoint>
    <timepoint id="t3">everything complete</timepoint>
  </timepoints>
  <activities>
    <activity id="u1">build</activity>
    <activity id="u2">assemble</activity>
```

```
    <activity id="u3">test</activity>
  </activities>
  <occurrences>
    <occurrence activity="u1" id="p1">build automobile
engine</occurrence>
    <occurrence activity="u2" id="p2">assemble engine</occurrence>
    <occurrence activity="u3" id="p3">test engine</occurrence>
  </occurrences>
</psl>
```

Here is the Activity Occurrences architectural document generated by the architecture engine. Most of the IDEF3 markup and content is suppressed. All that remains from the IDEF3 document are the timepoints and IDEF3 processes, which have become activity–occurrences in PSL. The `uob` and `id` attributes of `process` are suppressed because they have no counterpart in the Activity Occurrences DTD.

```
<?xml version="1.0"?>
<!DOCTYPE occur SYSTEM "actoccur.dtd">
<occur>
  <timepoints>
    <timepoint id="t1">start</timepoint>
    <timepoint id="t2">assembly complete</timepoint>
    <timepoint id="t3">everything complete</timepoint>
  </timepoints>
  <occurrences>
    <occurrence beginof="t1" endof="t3">build automobile
engine</occurrence>
    <occurrence beginof="t1" endof="t2">assemble engine</occurrence>
    <occurrence beginof="t2" endof="t3">test engine</occurrence>
  </occurrences>
</occur>
```

Benefits and Practical Considerations

We have shown that architectural forms in an XML presentation of a process specification can be used to:

- Generate a PSL extension–specific view of the data. This is useful for situations where the data is to be processed by an application supporting some PSL extensions but not others.
- Convert non–PSL terms into their corresponding PSL counterparts.

These capabilities of architectural forms benefit the PSL project in several ways:

- Process specification terminology can easily be altered to accommodate the preferences of different classes of users.
- Through judicious use of architectural forms, PSL DTDs can be extended without breaking existing applications.
- If dependencies between PSL modules are documented using architectural forms, reusability of processing software can be maximized. For instance, application software developed for the Activity Occurrences extension can be reused for other PSL extensions

that depend on the Activity Occurrences ontology.

In order to truly realize the benefits of architectural forms, the PSL project must take some practical considerations into account. One issue is the lack of maturity of XML and in particular XML software for processing architectural forms. Because the architectural form software tools available for XML as of this writing (Summer of 1999) are of beta quality, the automobile engine example from the previous section was originally written in SGML, tested using an SGML architecture engine, and then converted to XML. We expect the quality of XML architectural form processing software to eventually improve, but in the meantime such workarounds will be necessary.

We also need to create DTDs for more PSL extensions and improve our methodology for mapping PSL concepts to XML. The PSL Core and Activity Occurrences DTDs in the previous section's example were developed in an ad–hoc manner without a well–defined set of principles. The ideas stated in the section on "PSL and XML" should be expanded and formulated into a set of guidelines for representing PSL as XML. Perhaps these guidelines should become part of the future PSL standard.

Architectural Forms and Modularization

There is a growing realization among developers of frameworks for data exchange that modularity and extensibility are essential. Past experience tells us that large monolithic specifications are not flexible enough to meet the needs of their intended users and, as a result, application developers end up adding their own ad–hoc customizations. These modifications complicate data sharing and cause software maintenance problems when specifications change. PSL's modular approach heeds these lessons from the past. On the other hand, it introduces the challenge of maintaining consistency between and effectively using the PSL Core and extensions. Architectural forms help address this challenge.

Other projects besides PSL are using architectural forms to help achieve the benefits of modularization. Gary Simons at the Summer Institute of Linguistics used architectural forms to develop a "lean and mean" task–centered XML application derived from the larger and far more general TEI (Text Encoding Initiative) DTD [SIM 98]. Simons modified the TEI DTD's SGML declaration to make it compatible with XML, used architectural form attributes to map element names from the XML DTD to their TEI equivalents, and specified a TEI base architecture in his DTD. His XML application achieves TEI conformance but without all the excess baggage associated with the TEI DTD.

Architectures are also being used in a multi–level framework proposed by the HL7 SGML/XML Special Interest Group for creating, sharing, and processing electronic healthcare records [HL7 98]. The levels in the framework represent varying degrees of markup granularity and specificity, with the framework specifying an architectural DTD for each level. Level 1, the least granular level, represents information identifying and classifying the document, event, patient, and practitioner. Level 2 adds additional detail by structuring the document body into sections. Level 3, the most granular and most detailed level, aims to meet the processing requirements for a full electronic patient record.

Both of the preceding projects, like PSL, are making a conscious effort to promote extensibility and flexibility while avoiding the mistakes of past approaches. Architectures provide a powerful tool for achieving these goals. Architectural forms provide the rigor needed to clearly and unambiguously document the relationships between the specifications in today's modular environments. That is to say, architectural forms safeguard against "messing with Mr. In–between."

Acknowledgements

This project is funded by NIST's SIMA (Systems Integration for Manufacturing Applications) Program [FOW 99].

We would like to thank our colleagues at NIST as well as the Markup Technologies '99 peer reviewers for their helpful feedback and suggestions for improving an earlier draft of our paper. We are particularly grateful to Don Libes for his meticulous review and thoughtful comments.

Bibliography

[FOW 99] Fowler, J., *Systems Integration for Manufacturing Applications Program 1998 Annual Report*, NISTIR 6339, NIST, Gaithersburg, MD, April 1999.

[GEN 92] Genesereth, M., Fikes, R., *Knowledge Interchange Format (Version 3.0) – Reference Manual*, Computer Science Department, Stanford University, 1992.

[GOL 92] Charles Goldfarb, *The SGML Handbook*, Oxford University Press, 1992.

[HL7 98] HL7 SGML/XML SIG, *HL7 Patient Record Architecture*, DRAFT – September 4, 1998, amended – February 17, 1999.

[IDEF] IDEF Users Group, Working Group 1, *IDEF−U.G: the IDEF framework for methodology*, Version 1.0, Draft, Document IDEF–U.G.–0001, 1990.

[IDEF3] *IDEF3 Method Report*, IDEF Methods Web site, Knowledge Based Systems, Inc., http://www.idef.com.

[ISO 8879] ISO 8879:1986(E), *Information processing—Text and Office Systems—Standard Generalized Markup Language (SGML) − First Edition*

[ISO 10744] ISO/IEC 10744:1997, *Information processing—Hypermedia/Time−based Structuring Language (HyTime) − 2d edition*, A.3 Architectural Form Definition Requirements (AFDR).

[ISO 10744−1] ISO/IEC JTC 1/WG4, *ISO/IEC 10744 Amendment 1*, ISO/IEC JTC 1/WG4 N1957, 4 December 1997, http://www.ornl.gov/sgml/wg4/document/1957.htm.

[KIM 97] W. Eliot Kimber, *A Tutorial Introduction to SGML Architectures*, ISOGEN International Corporation, 1997, http://www.isogen.com/papers/archintro.html.

[MEG 98] David Megginson, *Structuring XML Documents*, Prentice Hall, 1998, Chapters 9–11.

[MER 44] Johnny Mercer (American lyricist), *Ac−Cent−Tchu−Ate the Positive*, E.H.

Morris, 1944.

[POL 98] Polyak, S., Aitken, S., *Manufacturing Process Interoperability Scenario*, AIAI–PR–86, Artificial Intelligence Applications Institute (AIAI), Edinburgh, 1998.

[PSL] Process Specification Language Web site, National Institute of Standards and Technology, http://www.nist.gov/psl/.

[SCH 99−1] Schlenoff, C., Gruninger, M., Ciocoiu, M., *The Essence of the Process Specification Language*, to appear in the Special issue on Modeling and Simulation of Manufacturing Systems in the *Transactions of the Society for Computer Simulation International*, late 1999.

[SCH 99−2] Schlenoff, C., Gruninger M., Tissot, F., *The Process Specification Language (PSL): Version 1.0*, to appear as a NIST Interagency Report (NISTIR) in mid–1999.

[SIM 98] Gary F. Simons, *Using architectural processing to derive small, problem−specific XML applications from large, widely−used SGML applications*, proceedings of *Markup Technologies '98*, Chicago, November 1998.

[W3C 98] World Wide Web Consortium, *Extensible Markup Language (XML) 1.0*, W3C Recommendation, 10–February–1998.

[W3C 99−1] World Wide Web Consortium, *XML Schema Part 1: Structures*, W3C Working Draft, 24 September 1999.

[W3C 99−2] World Wide Web Consortium, *XML Schema Part 2: Datatypes*, W3C Working Draft, 24 September 1999.

Biography

Joshua Lubell
National Institute of Standards and Technology
Manufacturing Systems Integration Division
100 Bureau Drive, Stop 8260
Gaithersburg MD 20899–8260 USA
E–mail: lubell@nist.gov

Joshua Lubell designs and implements software systems for information–based manufacturing applications at the National Institute of Standards and Technology in Gaithersburg, Maryland, USA. His technical interests include SGML and XML, database technology, and Internet computing. His previous experience includes artificial intelligence systems design and prototyping, software development for the building materials industry, and knowledge engineering. He has an M.S. in computer science from the University of Maryland at College Park, where he performed graduate research in diagnostic problem solving, and a B.S. in mathematics from the State University of New York at Binghamton.

Craig Schlenoff
National Institute of Standards and Technology

Manufacturing Systems Integration Division
100 Bureau Drive, Stop 8260
Gaithersburg MD 20899−8260 USA
E−mail: craig.schlenoff@nist.gov

Craig Schlenoff is a mechanical engineer and the program manager of the Process Engineering Program in the Manufacturing Systems Integration Division at the National Institute of Standards and Technology. This Process Engineering Program's goal is to develop standard representations of process to allow companies to precisely document their manufacturing processes and share these processes among different functions both within their organization and among partnering organizations. He is also the principal investigator of two projects within this program. The first, entitled the Process Specification Language Project, is defining a neutral representation (a language) for manufacturing processes which could be used for the sharing of process information among all functions in which it may be used. The second, entitled Ontologies for Interoperabilities, is applying the principle of ontological engineering to manufacturing systems integration with the output being an ontology/taxonomy of manufacturing concepts and functions to provide a common, shared vocabulary of terms and meanings for manufacturing systems integrators. Mr. Schlenoff received an M.S. in mechanical engineering from RPI (Rensselaer Polytechnic Institute) and a B.S. in mechanical engineering from the University of Maryland at College Park.