From XML to ABAP Data Structures and Back: Bridging the Gap with XSLT

Karsten Bohlmann, Christian Fecht, and Christoph Wedler

Karsten Bohlmann, Server Technology Group, SAP AG

Christian Fecht, Business Programming Languages Group, SAP AG

Christoph Wedler, Business Programming Languages Group, SAP AG

(complete bios appear on page 51)

XML, the EXtensible Markup Language, is the web standard for generic representation of structured data. Perhaps it hasn't yet found its way into your company's ABAP application development, but there is a fair chance that it will. Over the past few years, XML has left the hype phase and has developed into the central standard for open application integration — which is a pressing concern for most SAP customers.

The key to XML's immense success is that it has no meaning. Bare XML is just a generic model of trees, together with a generic syntax ("markup") for writing down elements of this model. By restricting the names of tree nodes and the admissible tree structures (in XML jargon, defining a "schema"), arbitrary subsets of the general model can be identified. Only by defining an application with a certain behavior on such a subset do you get something useful, such as a message format, a layout vocabulary, or even a programming language. The potential for creating new schemas or extending existing ones is unlimited. Different XML "dialects" can be combined; for example, a fragment in some layout dialect might be incorporated in a message of another dialect. XML namespaces provide the necessary modularization mechanism. It's this ability to compose what you like at will that makes XML such an appealing standard for application integration.

For many ABAP developers, mention of XML conjures up images of the *SAP Business Connector* (BC). The BC offers XML-based communication for SAP systems via a *remote procedure call* (RPC) mechanism. That is, the BC enables you to call R/3 interface functions from external software components and vice versa, using the XML standard to pass data back and forth in these calls. The use of XML, rather than

some proprietary binary or text format (as in more antiquated integration mechanisms), is considered an advantage because it lowers the hurdle for non-SAP software.

It would be a mistake, however, to think that all communication must be some kind of RPC. Certainly with XML as a data communication format, this is too narrow a view. In the SAP Web Application Server, there are many ways in which you can get hold of XML. The most common method is to send and receive XML messages via HTTP. You can also piggyback XML documents on one of the existing RPC mechanisms. Or you can have Business Server Pages (BSPs) that produce XML.

This article is about handling XML data, coming from any of these (or other) sources, in ABAP programs. To make the difference between the methods discussed in this article and XML-based RPC completely clear, think of what you would usually do with the BC: In order to wire up an external program — e.g., written in Java — with an R/3 system, you would employ the BC to call some BAPIs in R/3 or to call some external functions out of R/3. The BC would wrap the data of the calls in a special XML format, which is then translated into the appropriate data structures on either side of the connection. The BC would also take care of other technical aspects of the communication, such as session management.

In contrast, the tools we'll talk about here are not concerned with details of the communication mechanism at all, but only with XML as the format of the data content. You can't use these tools to set up a connection with anything, but if you do have a connection on which you want to speak XML, they will give you a smooth mapping between XML and ABAP structures. For example, suppose there is an information provider that offers a service (say, flight information) in the form of XML messages via HTTP. You can connect your ABAP application to this service via HTTP (a subject *not* treated in this article) and decode the messages into your own data structures without having to struggle with the XML documents in ABAP. Conversely, if you want to offer such a service yourself, you can define arbitrary mappings from ABAP data structures to XML. Thus the XML handling discussed here offers both more and less than XML-RPC: more, because it is completely general, and less, because it does not cover the actual communication.[1]

The tool we use to establish mappings between XML documents and ABAP data structures is the W3C standard XSLT (EXtensible Stylesheet Language Transformations). It is a declarative tree transformation language that was developed explicitly for the purpose of dealing with the ever-growing mass of XML schemas. In an XSLT program, you define a set of rules that express how an XML input tree is transformed into an XML output tree. In the 6.10 release of the kernel, SAP has implemented an XSLT processor to support this language and as an innovation, it has extended the application area from XML to ABAP data structures. That means you can write XSLT programs that do transformations between XML documents and data structures. This feature was expanded with the 6.20 kernel release, and it is the main focus of the present article.

After a general discussion of XML as a data model and XSLT as a transformation language, we delve into the basics of this new tool. A detailed introduction is beyond the scope of this article; therefore, some basic knowledge of XML is required and some prior knowledge of XSLT and XPath is useful. You will learn how to create and call XSLT programs, and how your programs can use XSLT to interface XML from ABAP in a way that allows them to retain given data structures on the ABAP side, while supporting arbitrary schemas on the XML side. Some more SAP-specific extensions, such as ABAP calls from XSLT and the "classical" use of XSLT as a stylesheet language (with resulting HTML), are covered in brief.

---

[1] As a matter of fact, the BC also offers the feature of *mapping* between external XML formats and its own "RPC data" format. If you are mainly interested in such a type of mapping, then you may find a better way to do the job in this article.

Background on XML and XSLT

SAP's overriding goal for the new Web Application Server[2] architecture is the support of open web standards. The move away from proprietary formats and mechanisms is founded on the conviction that cross-systems integration is to be a key feature of every successful software product. It will then no longer be an individual vendor's decision which standards are "good" and should be supported. "Good" becomes synonymous with "generally used." With this definition, XML is an extremely good standard. This holds even though you may complain that its syntax is clumsy, its data model incomplete, and its physical structuring mechanism a mess: it will dominate the future of web applications anyway. As remarked previously, its success is mainly due to the fact that it provides a unified mechanism for the definition of tree-shaped data schemas, without saying anything about their semantics, but allowing them to be composed arbitrarily.

XSLT, then, is a programming language that is defined both *in* XML and *for* XML: its concrete syntax follows an XML schema and its purpose is to *transform* XML trees. XSLT is emerging as an important technology as a direct consequence of the openness of XML: since XML alone offers interoperability on a purely syntactic level and allows for an open-ended set of schemas with individual semantics, there is an urgent need for a technology that supports the mapping between instances of related, but distinct schemas.

This need was recognized by the W3C during the design of a document-formatting language for XML

[2] The SAP Web Application Server is defined as a combination of the web-enhanced "classic" SAP server technology and a J2EE server. This article is exclusively concerned with the former — i.e., with ABAP-based development. The Java component has XML support, too, but is not discussed here.

Why This XML Syntax?

A common reaction upon first sight of an XSLT program is: "What an ugly syntax." The keywords of the language are XML elements themselves and they are distinguished from non-keywords by being in a certain namespace. This is not really a way of defining a syntax that will win awards for elegance, but it has some advantages. In fact, XSLT is a good example of the *composability* of XML. The purpose of an XSLT program is to produce a result of a certain schema. It is most natural to write down the elements of that resulting schema literally in the program. But then, in order to get well-formed XML, the instructions of the transformation language must be in XML format as well. The method to distinguish them from the other elements is to put them in their own namespace. This is illustrated in the following XSLT fragment:

```
<xsl:if test="@name">                <!-- if current node has a "name" attribute... -->
  <customer-name>                    <!-- create a "customer-name" result element... -->
  <xsl:value-of select="@name"/>     <!-- with the attribute value as content... -->
  </customer-name>                   <!-- and close the result element -->
</xsl:if>                            <!-- end of conditional fragment -->
```

So, XSLT can be composed with any other schema. And, since XSLT programs are well-formed XML, they can even be processed by XSLT. (In fact, the SAP XSLT compiler is partly written in XSLT itself.)

called XSL, which led to the separation of the transformation-related part from the layout-oriented part of the language, and to the tongue-twister name "XSLT." A simple, but powerful expression language, *XPath*, for navigating XML trees was specified separately and incorporated into XSLT.[3]

The purpose that the XSLT-XPath combination serves has been aptly characterized as "the SQL of the web" — just as SQL "navigates" and "constructs" elements of the relational data model (i.e., tables), XSLT-XPath operates on the XML tree model. The distinguishing features of the language are that it is:

- **High-level:** The concept of tree transformation (as a combination of tree navigation and construction) is fundamental to the language; programmers don't have to build transformations indirectly from lower-level concepts.
- **Declarative:** Transformations are expressed functionally, without manipulation of an explicit state (say *what* you want to do, not *how* you want to do it).
- **Rule-based:** Flow of control is made implicit by pattern-matching against the source tree.

The typical advantages associated with high-level, declarative languages hold for XSLT (as compared to XML handling in procedural languages):

- Higher development productivity due to shorter programs and debugging times, and
- Better modularity due to rule-based, side-effect-free flow of control, leading to
- Better maintainability and adaptability, especially in connection with schema evolution on the input or output side, which is frequent in XML.

> ✓ ***Read More About It***
>
> *While a thorough introduction to XSLT and XPath is beyond the scope of this article, we encourage you to pursue this interesting topic. Recommended resources for learning more are listed in the sidebar "Recommended Reading" on page 49.*

The core XML services on which many existing SAP applications are already based are provided by the iXML[4] library, which has been around since Release 4.6D. It contains a *parser*, a *renderer*, and a *DOM implementation*[5]. These tools have been implemented in C++ in the kernel and have been made accessible from ABAP by means of proxy classes, i.e., ABAP classes whose implementation consists of kernel calls. You can find extensive documentation for the iXML library in SAP's online help (**http://help.sap.com**) under **SAP Library→ SAP Web Application Server→ Basis Services / Communication Interfaces → XML Library**.

In kernel release 6.10, an *XSLT processor* — an engine that executes XSLT programs — was added. There were several motivations for SAP to do this. One of them is the classical "stylesheet" use of XSLT: the result of an XSL transformation can be XML, but also HTML, WML, or any other text-based document format, so the combination of XML and XSLT can be used to implement device-dependent

[3] The layout-oriented part is called XSL-FO (Formatting Objects), so the equation is: XSL = XSLT + XSL-FO. The formatting schema XSL-FO is not addressed by the SAP Web Application Server. XSLT (**www.w3.org/TR/xslt**) and XPath (**www.w3.org/TR/xpath**) have the status of W3C recommendations; the current version is 1.0 for both (dating from November 1999). Versions 2.0 are under way.

[4] Just to prevent mystification, the "i" stands for Integrated (into the kernel).

[5] The DOM (Document Object Model) is an object-oriented programming interface to (a slight variation of) the so-called "XML Infoset," the abstract data model of XML. Variations of the model are due to inefficiencies in the evolution of XML standards. The XPath data model (which is also used by XSLT) is very similar, but not identical, to the XML Infoset and the DOM. The iXML library supports the DOM Level 2 core specification and an event-based parsing interface similar to SAX (Simple API for XML).

user interfaces. We will briefly illustrate this use at the end of the article. Even more important in the SAP context is the use of XSLT for XML-to-XML transformations, for example, in "middleware" type applications. And finally, a further application area of XSLT was created as a genuine SAP specialty: the mapping between XML and ABAP data structures, which we'll examine next.

XML, ABAP, and XSLT: Two Data Models and a Bridge

We have emphasized the role of XML as a data model, which, not only to SAP, has turned out to be far more interesting than its original designation as a document markup language. A more classical habitat of data models is, of course, programming languages. While the data model of a programming language, like ABAP, is primarily the basis for data *processing*, the strength of XML is data *communication*. The need to link these two data models arises naturally when ABAP-based systems are integrated via XML.

An XML Format for ABAP Data Structures

Let us picture an integration scenario where we have some data in ABAP that we must communicate to another system or application as XML. Or, conversely, we want the ABAP data to be filled from an XML document that we receive. We will ignore objects for the moment, so our data consists of basic ABAP types, structures, and internal tables.

To make it concrete, let's take the good old "SFLIGHT" example known from countless ABAP courses. Say we have an internal table of the structured type `BAPISFLDAT`. A value of this type, containing information for a single flight, might look like **Figure 1**.

This is not hard to express in XML. Just about any structure can be encoded in XML, and particularly well if it is tree-shaped, such as a cycle-free data

Figure 1 Internal Table of Type BAPISFLDAT

Component	Content	Type	
AIRLINEID	DL	C	3
AIRLINE	Delta Airlines	C	20
CONNECTID	1699	N	4
FLIGHTDATE	20011123	D	8
AIRPORTFR	JFK	C	3
CITYFROM	NEW YORK	C	20
AIRPORTTO	SFO	C	3
CITYTO	SAN FRANCISCO	C	20
DEPTIME	171500	T	6
ARRTIME	203700	T	6
ARRDATE	20011123	D	8
PRICE	513.6900	P	12
CURR	USD	C	5
CURR_ISO	USD	C	3

Listing 1: A BAPISFLDAT Value in asXML

```
<BAPISFLDAT>
  <AIRLINEID>DL</AIRLINEID>
  <AIRLINE>Delta Airlines</AIRLINE>
  <CONNECTID>1699</CONNECTID>
  <FLIGHTDATE>2001-11-23</FLIGHTDATE>
  <AIRPORTFR>JFK</AIRPORTFR>
  <CITYFROM>NEW YORK</CITYFROM>
  <AIRPORTTO>SFO</AIRPORTTO>
  <CITYTO>SAN FRANCISCO</CITYTO>
  <DEPTIME>17:15:00</DEPTIME>
  <ARRTIME>20:37:00</ARRTIME>
  <ARRDATE>2001-11-23</ARRDATE>
  <PRICE>513.69</PRICE>
  <CURR>USD</CURR>
  <CURR_ISO>USD</CURR_ISO>
</BAPISFLDAT>
```

structure. **Listing 1** shows the same flight information as a fragment of the *ABAP Serialization XML* format (*asXML*) for ABAP data structures.[6]

[6] As a matter of fact, the *order of structure components* in asXML is irrelevant. For instance, <AIRLINE> could be put before <AIRLINEID>. But since the XML tree model lacks the concept of structured, unordered nodes (it has only structured, ordered nodes ["elements"] and unstructured, unordered nodes ["attributes"]), some redundant order must be added.

Listing 2: An Internal Table of BAPISFLDAT in asXML

```
<asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  <asx:values>
    <FLIGHT_LIST>
      <BAPISFLDAT>  ...  </BAPISFLDAT>
      ...
      <BAPISFLDAT>  ...  </BAPISFLDAT>
    </FLIGHT_LIST>
  </asx:values>
</asx:abap>
```

This format was introduced under the name "XRFC" by the SAP Business Connector for the representation of parameters when calling function modules via XML. The example illustrates structures and simple values of some basic types. A structured value is represented as an element containing child elements, which are named according to its components. The representation of simple values conforms to the canonical representation of XML schema data types.[7] An internal table of flights would be represented as in **Listing 2**. A more detailed description of asXML may be found in the appendix to this article. For the moment, it suffices to observe that the two outer elements form a constant "envelope"; inside this envelope there is a list of symbolic names (here it's just one: `FLIGHT_LIST`) and under each name there is the XML representation of an ABAP value.

As part of the SAP Web Application Server development, asXML was implemented in the kernel. Since Release 6.10, it is possible to convert the XML representation to the ABAP value. Since Release 6.20, the reverse direction is possible as well. The conversion is usually referred to as the *identity transformation* because it transforms a tree from one data model to the other without changing its structure.

So, are we finished? With the identity transformation, we can represent any ABAP data structure in XML and send it to another system, and we can prescribe a format for any kind of information that we wish to receive as XML in order to fill a data structure. Probably we will not always expose our data structures directly for XML communication, but will have a restricted set of "communication data structures" and implement transformations (in ABAP) between these structures and the actual source or target structures. We refer to this approach as *inside-out*, or *ABAP-centric*.

Of course, prescribing formats inside-out is not the way things work in the world of open standards. If we can understand only XML that obeys the "meta-schema" derived from the ABAP type system, that buys us little or nothing. For instance, a worldwide standard "XFlight" for flight information might emerge that looks like what is shown in **Listing 3**. So, if an inside-out approach is not the answer, what is the alternative?

Dealing with External XML Formats

The opposite of the inside-out method sketched in Listing 3 is obviously *outside-in*, which would mean that we become XML-centric rather than ABAP-centric. We would support arbitrary XML schemas by deriving, for each schema, a data structure that reflects precisely the XML structure and then use the identity transformation again to move the data between the two models. It's not hard to imagine what the data structure derived from XFlight would look like.

[7] Which has also been adopted by the SOAP (Simple Object Access Protocol) standard.

Listing 3: Hypothetical Standard XML Format "XFlight"

```
<connections>
 <connection city-from="NEW YORK" city-to="SAN FRANCISCO" from="JFK" to="SFO">
   <carrier code="DL" name="Delta Airlines">
     <flight code="1699">
       <departure>
         <date>2001-11-23</date> <time>17:15:00</time>
       </departure>
       <arrival>
         <date>2001-11-23</date> <time>20:37:00</time>
       </arrival>
       <price>
         <amount>513.69</amount>
         <currency>USD</currency> <currency-iso>USD</currency-iso>
       </price>
     </flight>
   </carrier>
   <carrier code="AA" name="American Airlines">
     ...
   </carrier>
 </connection>
</connections>
```

So, is the outside-in approach then the answer? There are several activities in the Java community that follow this approach, most prominently JAXB[8]. The concept is known as *data binding*. There are at least four good reasons *not* to adopt it:

1. The class of XML schemas that can be sensibly mapped to data types is restricted. In particular, a schema has to be highly regular. Irregular structures can be expressed only generically, and very inefficiently, as graphs of objects.

   For Java, this argument is not so clear because Java suffers from the "objects-only" restriction. It wouldn't make sense to adopt this restriction for ABAP-XML binding by providing poor support for structures and tables.

2. The approach results in an inflation of data types.

3. Existing functionality cannot work on these new data types. ABAP adaptation code would have to be written to map between old and new types.

4. A good data *communication* structure is not, in general, a good data *processing* structure. Thus, even if no existing functionality has to be adapted to the schema-derived data structures, there is a fair chance that new functionality built around these structures will be badly designed.

Well, if it's bad to be ABAP-centric, and it's bad to be XML-centric, what's left? What we need is a *symmetric* solution for ABAP-XML exchange that:

- Supports arbitrary ABAP data structures without inflating the set of XML schemas
- Supports arbitrary XML schemas without inflating the set of data types
- Does not require ABAP code to map between processing and communication structures

[8] JAXB is being developed through the Java Community Process℠ program under JSR-31.

Listing 4: XSLT Program That Converts XFlight to asXML

```
<xsl:transform  version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/
                                          Transform">

<xsl:strip-space elements="*"/>

<xsl:template match="connections">
  <asx:abap version="1.0" xmlns:asx="http://www.sap.com/abapxml">
    <asx:values>
      <FLIGHT_LIST>
        <xsl:apply-templates/>
      </FLIGHT_LIST>
    </asx:values>
  </asx:abap>
</xsl:template>

<xsl:template match="connection">
  <xsl:variable name="conn" select="."/>
  <xsl:for-each select="carrier">
    <xsl:variable name="carr" select="."/>
    <xsl:for-each select="flight">
      <xsl:variable name="dep"   select="departure"/>
```

- Does not encourage flawed application design based on the XML data model

This is precisely what the SAP XSLT implementation provides.

Transforming Trees with XSLT

The solution is to take *both* the asXML format for ABAP data structures *and* arbitrary external XML formats, and to use the XML transformation language XSLT as a bridge to define the necessary mappings between them. **Listing 4** shows an XSLT program that transforms XFlight into the internal `BAPISFLDAT` format.[9]

This is a very simple XSLT program; it just loops over the three structure levels of XFlight documents (`connection – carrier – flight`) and churns out the flights as a flat list of `BAPISFLDAT` records. XSLT instructions are recognizable by the `xsl:` namespace prefix; the rest of the code is literal result elements. The `xsl:for-each`, `xsl:variable`, and `xsl:value-of` instructions use XPath expressions to select sets of nodes from the source tree. For example, the expression `$conn/@from` selects the `from` attribute (a singleton node set) of the node bound to the variable `conn` (which is the current `connection` element).

Now it is time to reveal the truth about the "identity transformation" whose existence was asserted above. Conceptually, the mapping of an arbitrary XML tree to an ABAP tree by an XSLT program can indeed be understood as a (structure-changing) transformation from the XML tree to the canonical asXML tree, followed by the (structure-preserving) identity transformation that brings the asXML tree into the ABAP structure. But an optimization has been built

[9] You may note that, for example, dates are formatted as "2001-11-23" on the XML side, while they have to look like "20011123" on the ABAP side. In order to avoid every XSLT program having to perform string manipulations for simple values, the representation of these values is adapted automatically.

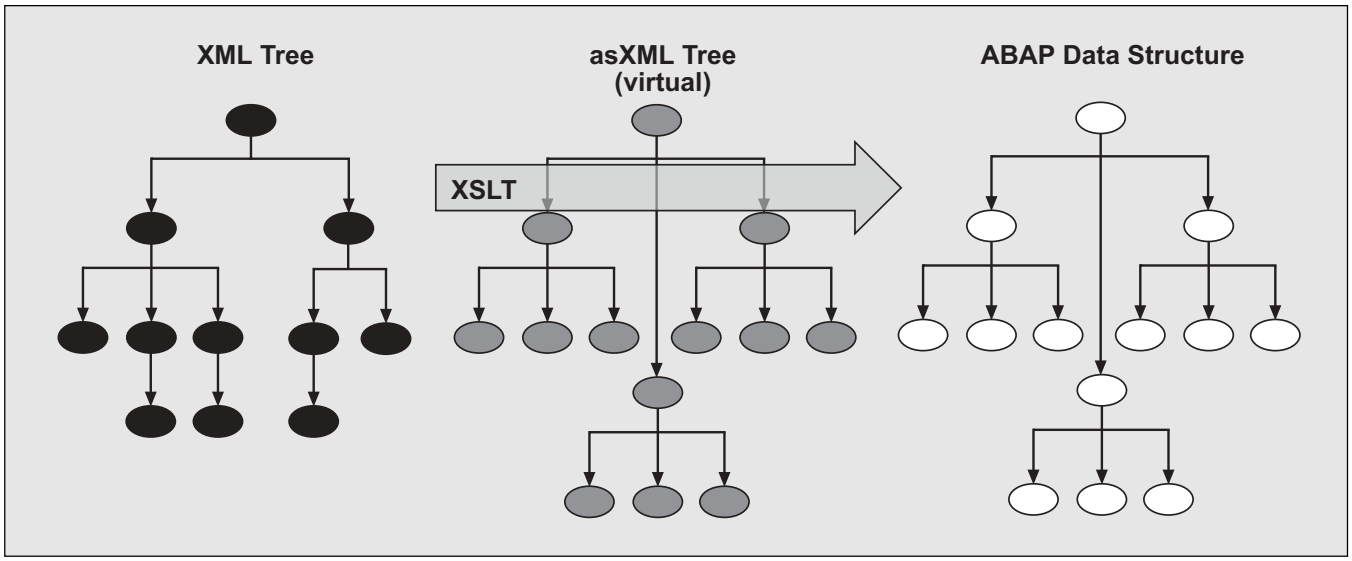

```
      <xsl:variable name="arr"   select="arrival"/>
      <xsl:variable name="price" select="price"/>
      <BAPISFLDAT>
        <AIRPORTFR ><xsl:value-of select="$conn/@from"/></AIRPORTFR>
        <CITYFROM  ><xsl:value-of select="$conn/@city-from"/></CITYFROM>
        <AIRPORTTO ><xsl:value-of select="$conn/@to"/></AIRPORTTO>
        <CITYTO    ><xsl:value-of select="$conn/@city-to"/></CITYTO>
        <AIRLINEID ><xsl:value-of select="$carr/@code"/></AIRLINEID>
        <AIRLINE   ><xsl:value-of select="$carr/@name"/></AIRLINE>
        <CONNECTID ><xsl:value-of select="@code"/></CONNECTID>
        <FLIGHTDATE><xsl:value-of select="$dep/date"/></FLIGHTDATE>
        <DEPTIME   ><xsl:value-of select="$dep/time"/></DEPTIME>
        <ARRDATE   ><xsl:value-of select="$arr/date"/></ARRDATE>
        <ARRTIME   ><xsl:value-of select="$arr/time"/></ARRTIME>
        <PRICE     ><xsl:value-of select="$price/amount"/></PRICE>
        <CURR      ><xsl:value-of select="$price/currency"/></CURR>
        <CURR_ISO  ><xsl:value-of select="$price/currency-iso"/></CURR_ISO>
      </BAPISFLDAT>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

</xsl:transform>
```

into the SAP XSLT processor — rather than actually constructing the intermediate asXML tree, the ABAP data structure is filled directly by the execution of the XSLT program. This process is depicted in **Figure 2**.

The reverse direction is analogous on the conceptual level. The ABAP data structure is brought into an asXML tree by the identity transformation and then an XSLT program transforms that tree into the

Figure 2 *Transformation from XML to ABAP Data*



resulting XML tree (or HTML, or text). But in this case, the intermediate asXML tree must indeed be constructed because XSLT programs require random access to the source tree through XPath "location path" expressions. Hence, the asXML tree in **Figure 3** is not just virtual — it is constructed by an implicit call of the identity transformation when the source tree is a data structure.

For our "XFlight" example, the direction from ABAP to XML is more demanding than the transformation of Listing 4 because a more complex structure is created in this direction. The corresponding XSLT program is shown in Listing 8 in the appendix (see page 54). It demonstrates that XSLT is not limited to simple translations, but can express complex structural transformations. In general, the result tree can be completely different from the source tree.

XSLT in the SAP Web Application Server

We have learned how the mapping between XML and ABAP structures via XSLT conceptually works. In order to apply these concepts, it is necessary to see how XSLT programs are actually maintained and invoked in the SAP Web Application Server. These topics are totally independent of the XML-ABAP mapping issue. You will follow the same procedures when you write XSLT programs to transform XML to HTML, for example.

Maintaining XSLT Programs in the Workbench

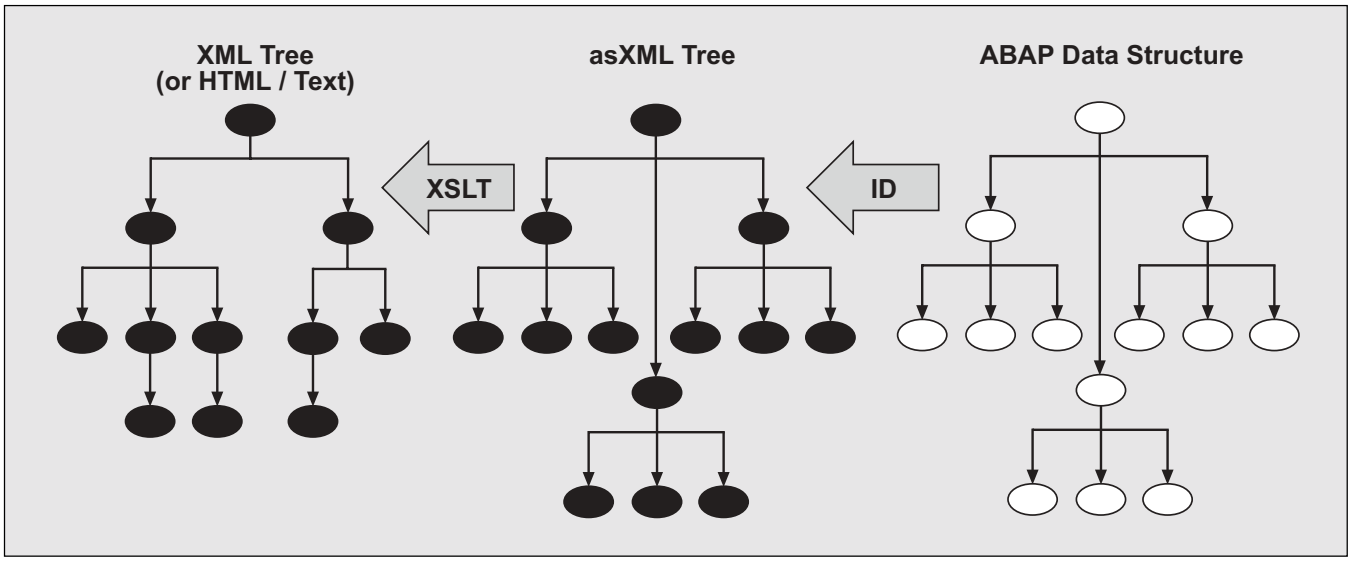
XSLT programs have been made *repository objects* in the SAP Web Application Server. If you are familiar with the SAP development environment, you know that this status implicates a number of highly integrated capabilities, ranging from creation, editing, and debugging to cross-system transport and delivery. In particular, you have the *activation* step known from ABAP programs and other development objects, in which a modified, inactive program version is checked for syntactic correctness and (if the check is successful) replaces the previous executable version.

A Behind-the-Scenes Look at How XSLT Has Been Implemented in the SAP Web Application Server

For a programming language implementation, the fact that it runs on an application server has important consequences. Most XSLT processors on the market follow an approach that is less than ideal in a server context. This approach can be characterized as "tree interpretation" — the XSLT program is parsed into a tree-like internal representation and this structure (possibly after some rewriting for optimization*) is interpreted recursively. Typically the stack of the implementation language (e.g., Java) is used for recursion, which has the drawback that program execution cannot easily be interrupted for purposes like process scheduling. Furthermore, there is no efficient way of storing and reloading the result of program analysis (the program tree). Therefore, it is efficient to execute one program many times in a row, but when the active set of programs grows too large, program trees must be discarded and then rebuilt when programs are needed again. On a serious application server, the number of programs tends to be very large and execution order tends to be random. Tree interpretation fails to scale under such conditions.

* This preprocessing step is sometimes (inadequately) termed "compilation."

Figure 3 Transformation from ABAP Data to XML



Creating Programs

A new XSLT program can be created by following these steps:

1. Start the Object Navigator (transaction SE80) and choose the "package" ("development class") view with the package in which you wish to create the program. Alternatively, choose your "Local Objects."
2. In the context menu (right mouse button) of the

These are exactly the same problems that arise for ABAP programs — and have been solved over the last 10 years by the SAP kernel in a manner that, as best as we can tell, remains unmatched by any other application server. The solution consists of a number of well-known principles, implemented with a high degree of sophistication. ABAP programs are not interpreted as program trees, but are translated to *byte code* that is interpreted on a *virtual machine*. The byte code (the "program load") can be stored to and retrieved from the database efficiently in the form of binary chunks. Elaborate compile-time *optimizations* can be afforded because their results are stored. When a program has been executed once, a subsequent invocation most likely does not even access the database, as the byte code will still be found in the *program buffer* PXA (Program eXecution Area).

This mature infrastructure has been reused for the SAP XSLT processor. Of course, since XSLT is an entirely different language, the ABAP virtual machine and byte code did not apply; a dedicated virtual machine was designed specifically for XSLT. The storage and buffering mechanisms, however, were taken over directly from the ABAP implementation, with the following main benefits for XSLT developers:

- ✓ **Physical storage:** You need not care about the physical storage of XSLT programs; you just edit and run them. XSLT sources and loads are stored transparently in the database.

(continued on next page)

root of the object tree, follow the path **Create** → **More...** → **XSL Transformation**.

3. In the "Create XSLT Program" dialog window, enter the name and a short description for the new program, and confirm your entries with . Also confirm the pop-up for the object-directory entry.

The name of the newly created program is displayed in the object list under the node "XSL Transformations." In the tool area of the Object Navigator you will see the XSLT tool. The "Properties" tab contains the short description, the name of the creator, the development class, and so on. By selecting the "Source" tab, you enter the source editor. A few lines of default content are already there, including the `<xsl:transform>` element with namespace declarations and an empty template rule matching the root node. This is where you will typically start editing.

Another way to create XSLT programs is to use the "Edit object" button () of the Object Navigator, which brings you to the "Object Selection" screen (**Figure 4**). With the "XSL Transformation" option located on the "More..." tab, you can not only create new programs, but also copy (), delete (), rename (), or navigate to (in "display" or "change" mode) existing programs.

Editing and Activating Programs

The XSLT editor is the same plain text editor that is used for ABAP sources. As a little editing aid, the "Tag Library" tree offers the ability to insert tags (XSLT, HTML, etc.) in the source text via drag&drop.

Just like in the ABAP editor, the XSLT editor includes "Check" () and "Activate" () buttons. The former triggers the XSLT compiler without making any change to the database, whereas the latter will update the program load if compilation is successful, making it public as the executable version. Compiler errors and warnings are displayed in a separate window at the bottom of the screen (**Figure 5**). A program containing syntax errors cannot be activated, but it can be saved () in an inactive state. As with all repository objects, *use* of an XSLT program — for example, inclusion by another XSLT program — always refers to the active version.

(continued from previous page)

- ✓ **Buffering:** XSLT loads go into the program buffer PXA for execution. The XSLT processor operates directly on the byte code in the PXA. The PXA's sophisticated buffering techniques take effect.
- ✓ **Sharing:** The PXA is allocated in the application server's shared memory, so program loads can be used by all the server's work processes simultaneously.
- ✓ **Synchronization:** If an XSLT source is changed on one application server, the change is propagated to all other servers. The PXA is informed that the current load version is outdated and that on the next request the new version has to be loaded.
- ✓ **Versioning:** The PXA supports multiple versions of the same XSLT program at the same time. If you change an XSLT program while a running transaction uses this program, then the execution continues with the old version whereas you will see the new version.

Figure 4 Selecting an XSLT Program

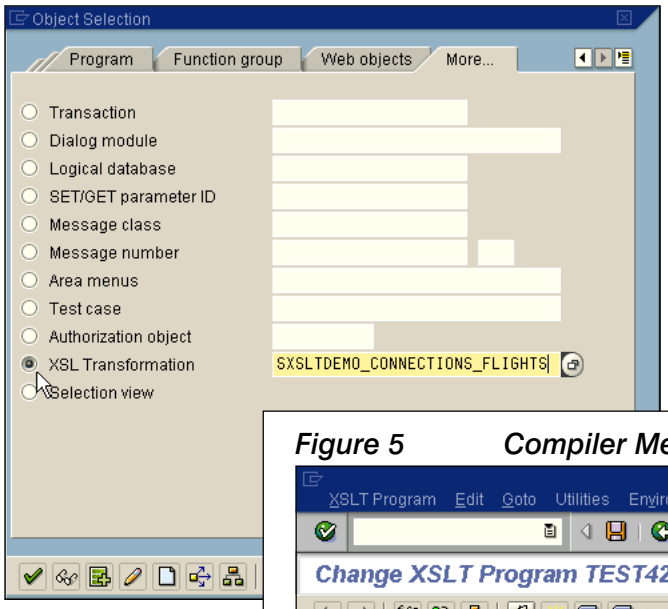
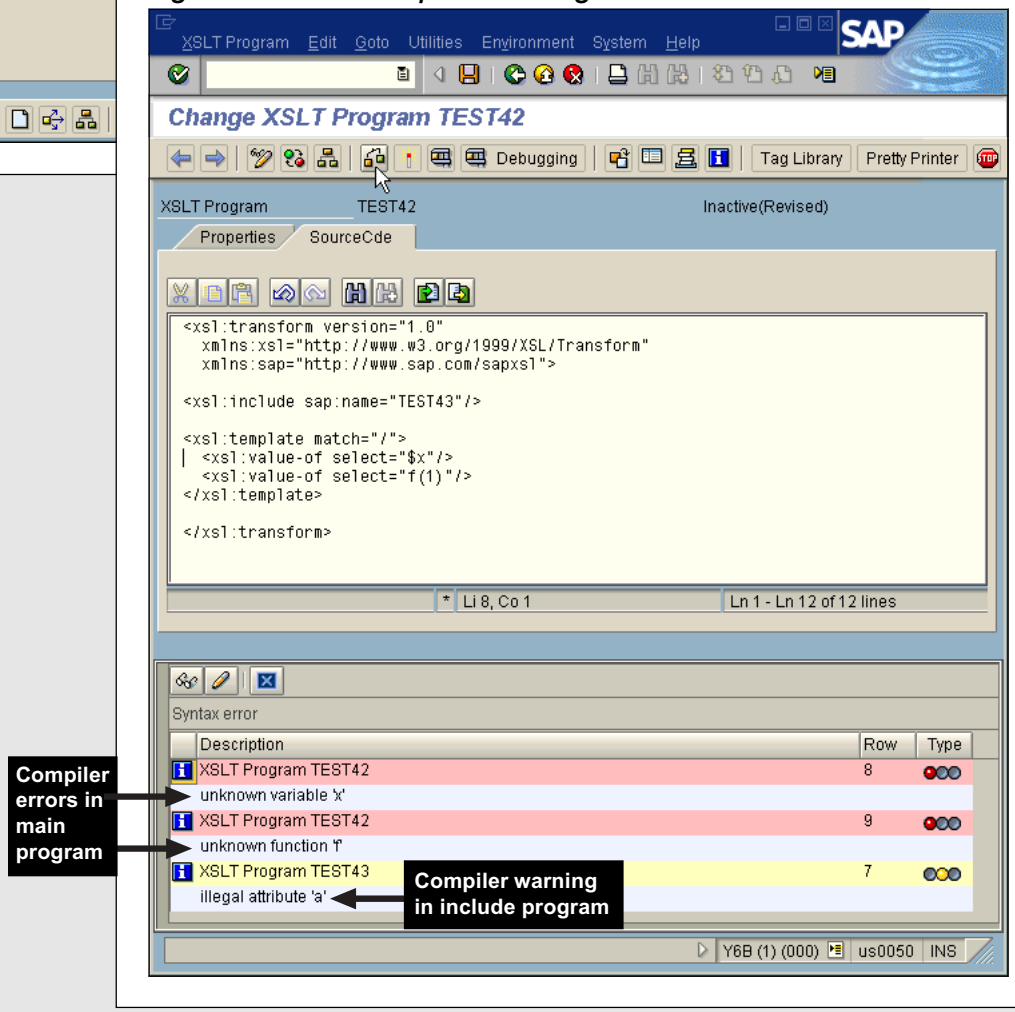


Figure 5 Compiler Messages in the XSLT Editor



Testing and Debugging Programs

You can test whether an active XSLT program does what it's supposed to do by supplying an XML input document, executing the program on it, and inspecting the result. You can do so directly from within the editor:

1. Choose to start the XSLT tester.
2. Specify the XML source document as a local file for upload.
3. Specify the result document as a local file for download, or choose to display it on the screen (either as a string or in the HTML Viewer).
4. You may supply (string-valued) program parameters.
5. Execute the program by selecting .

The visual XSLT debugger[10] allows you to track down errors, but also to better understand the functioning of your XSLT program. You can follow the execution step by step. In order to initiate debugging, set at least one breakpoint in the XSLT program you want to debug[11]:

1. In the editor, position the cursor on a line and click in order to set a breakpoint on this line. (Click again to remove it.)
2. Start the program either with the XSLT tester or from an auxiliary ABAP program. Execution will halt at the line where you set the breakpoint.

Rather than explain the debugger functions in detail, **Figure 6** gives a general impression. The XSLT program being debugged is displayed in the upper right window. Breakpoints are marked in yellow; the current line is colored blue. On the left there is the XML source tree, with the current node colored blue. The bottom window contains the result in its intermediate state. It may be switched to display the list of breakpoints, the call stack, or the values of variables. Execution can be continued in several modes, such as "single step," "to breakpoint," "to change of source node," and "to change of result" (the latter two are idiosyncratic of XSLT).

[10] The XSLT debugger is available with Release 6.20.

[11] There is not yet an option to "step into" an XSLT program, at the point of its invocation, from the ABAP debugger.

Invoking XSLT Programs from ABAP

After you have created an XSLT program in the workbench, you need a way to invoke it. Remember, XSLT is just a powerful language for tree transformations. It's not a general-purpose programming language. Therefore, you'll need to invoke XSLT programs from within your ABAP programs.

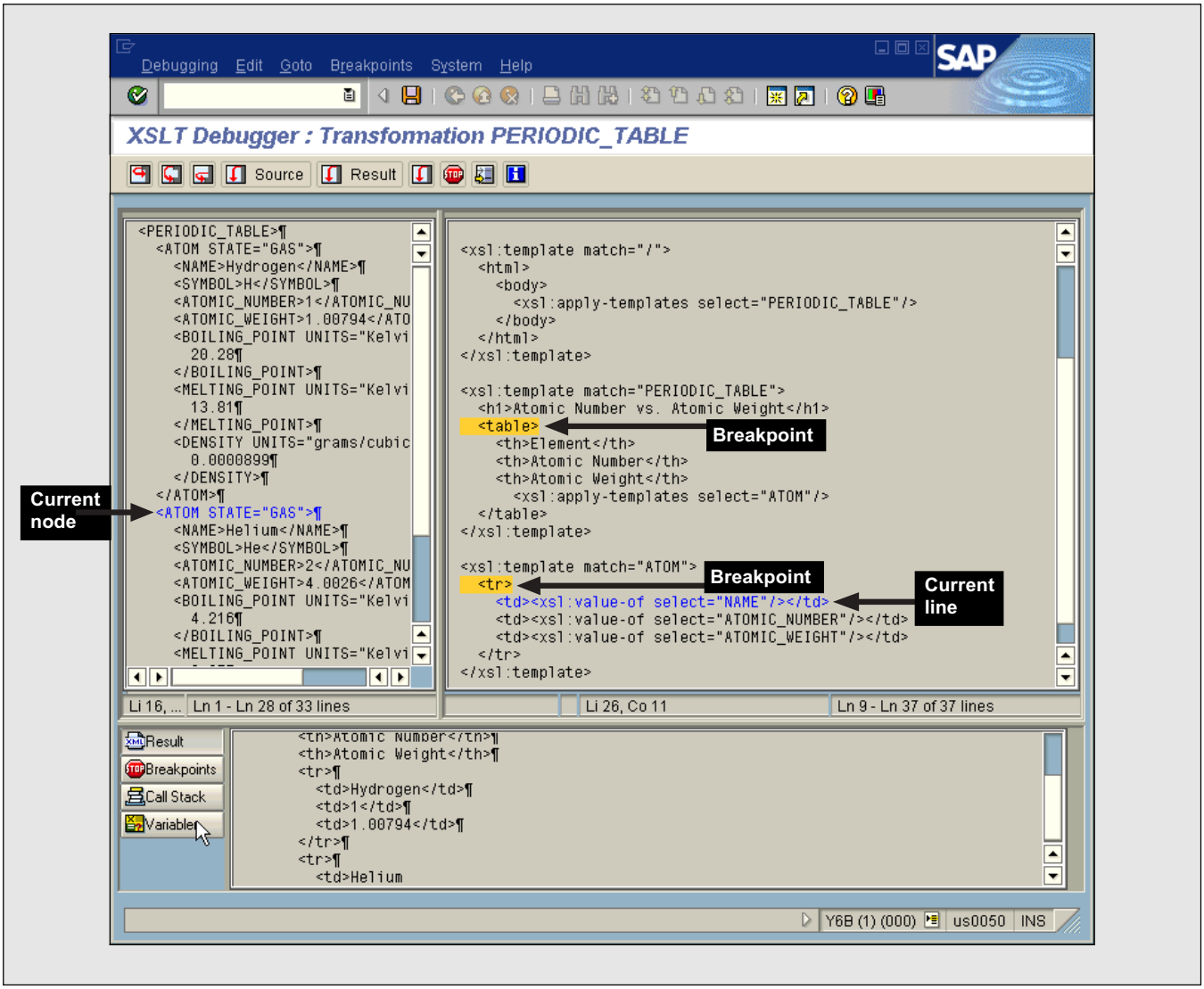
Toward this end, XSLT invocation has been implemented as a new ABAP statement `CALL TRANSFORMATION`, similar in nature to `CALL FUNCTION` and `CALL METHOD`. For example, the XSLT program shown back in Listing 4 (say it has been given the workbench name "CONNECTIONS") would be called as follows:

```
CALL TRANSFORMATION
       connections
     SOURCE XML
       xflight_string
     RESULT
       FLIGHT_LIST = it_sflight.
```

This statement activates the XSLT processor to transform the string `xflight_string` (with "XFlight" XML content) to an internal ABAP table `it_sflight` of `BAPISFLDAT` records. The statement's first component, `connections`, is the name of the XSLT program. The keywords `SOURCE` and `RESULT` introduce clauses that specify the source and result of the transformation. If a clause starts with the keyword `XML`, the tree is an XML tree; otherwise it is an ABAP tree.

An "ABAP tree" is specified by a list of bindings, `NAME = VAR`, where `VAR` is an ABAP variable in scope and `NAME` is the element name that encloses the XML representation of the ABAP value

Figure 6 *The XSLT Debugger*



(`FLIGHT_LIST` in Listing 2). A "binding name" serves as a level of indirection, one that decouples the XML representation from the concrete variable name.[12] The types of the variables used in the bindings determine the generated (ABAP source) or required (ABAP result) asXML representation.

`CALL TRANSFORMATION` can be used for XML-to-XML, XML-to-ABAP, ABAP-to-XML, and even ABAP-to-ABAP transformations. The XSLT program and the binding lists can be specified dynamically. An XML source or result can be provided in various ways, such as a string or a DOM node.

For example, in order to apply

- a transformation whose name is stored in the variable `xslt_program` to
- a table `bindings` (of type `abap_trans_srcbind_tab`) and obtain the result in

[12] Binding names introduced by `name = var` are converted to upper-case because of ABAP's case-insensitive nature. (With dynamically specified bindings, case-sensitive names are possible.) However, note that XML is case-sensitive, so binding names must be case-correct when they are matched or generated in XSLT programs.

- the DOM node `document` (of type `REF TO if_ixml_document`),

one writes:

```
CALL TRANSFORMATION
       (xslt_program)
     SOURCE
       (bindings)
     RESULT XML
       document.
```

Calling ABAP from XSLT, and Other SAP XSLT Extensions

As in all XML standards, the "X" in XSLT stands for eXtensible. The XSLT specification defines two legitimate ways for extending the language: *extension instructions* on the level of XSLT and *extension functions* on the level of XPath, which must be put into vendor-defined namespaces. The namespace chosen for SAP extensions is `"http://www.sap.com/sapxsl"`.[13]

One category of extensions consists of instructions and functions that add certain well-defined functionality. A typical example is an extension function for converting a value of type "result tree fragment" (an XML tree bound to a variable) to a value of type "node set," which overcomes a silly restriction of XSLT 1.0 and is offered by every self-respecting XSLT processor. In our case, it is called `sap:node-set()`.

Extensions of this kind require implementation in the XSLT processor. Another, very general way of extension can be opened up by implementing constructs for calls into *another programming language*. This may be in order to retrieve some additional information (from the database, for example) or to perform a computation that is not easily expressed in XSLT.

[13] Just like the XSLT namespace, the SAP-XSLT namespace must be declared by an XSLT program that uses SAP extensions, like this: `xmlns:sap="http://www.sap.com/sapxsl"`. (By convention, we use the prefix `sap`.)

✓ *Calling Out of XSLT: A Word of Caution*

One should be careful not to resort to the possibility of calling ABAP (or, in another context, another general-purpose language) from XSLT too readily. Quite a lot can be done by means of XSLT and XPath, including string and numeric computations and recursive functions. Mixing pure XSLT with ABAP needlessly tends to result in programs that are hard to understand and maintain. And, of course, side-effects through ABAP calls are to be avoided by all means, as they mix badly with the side-effect-free character of XSLT.

Only *method* calls to classes defined in ABAP Objects (but not function modules, etc.) are supported. This includes instance methods and (since Release 6.20) class methods and constructors. Instance methods and constructors, of course, raise the need to somehow treat instances of ABAP classes in XSLT. Since Release 6.20, these so-called *external objects* are a separate value type, like strings and numbers.[14] Hence, they can be bound to pre-existing ABAP Objects at the start of the XSLT program simply by declaring them as top-level parameters. Given an external object bound to variable "x," an instance method call then looks as shown in **Listing 5**.

The rather clumsy format is necessary to cover ABAP methods with more than one return value ("exporting" or "changing" parameters). This is captured by treating the language construct of an external call, from the XSLT perspective, as a *variable-binding* instruction. In this example, the value 42 is passed in the importing parameter `IMP` of the ABAP method `METHOD`, and the values returned in parameters `EXP1` and `EXP2` are bound to variables that can be accessed as "$v1" and "$v2" in successive XPath expressions. The variable scope for `sap:call-external` is just the same as for

[14] In Release 6.10, external objects were not a value type (i.e., on the same level as strings and numbers), but were declared as top-level entities, on the same level as template rules, using an extension instruction: `<sap:external-object name="x"/>`. The new interpretation of external objects is more general, but backward-compatible.

Listing 5: Calling ABAP from XSLT

```
<sap:call-external name="x" method="METHOD">
  <sap:callvalue param="IMP" select="42"/>
  <sap:callvariable name="v1" param="EXP1" type="boolean"/>
  <sap:callvariable name="v2" param="EXP2" type="number"/>
</sap:call-external>
```

Listing 6: Calling ABAP from XPath

```
<!-- top-level instance method declaration: -->
<sap:external-function name="p:f1" method="METHOD1" kind="instance">
  <sap:argument param="IMP1"/>
  <sap:argument param="IMP2"/>
  <sap:result param="EXP" type="number"/>
</sap:external-function>

<!-- top-level class method declaration: -->
<sap:external-function name="p:f2" class="CL_CLASS" method="METHOD2"
kind="class">
  <sap:argument param="IMP"/>
  <sap:result param="EXP" type="number"/>
</sap:external-function>

<!-- top-level constructor declaration: -->
<sap:external-function name="p:new" class="CL_CLASS" kind="constructor"/>

<!-- constructor, instance method, and class method call: -->
<xsl:value-of select="p:f1(p:new(), 42, 'foo') + p:f2('bar')"/>
```

`xsl:variable`. A hint for the expected XSLT type can be given so that, for example, a character "X" is not interpreted as a string, but as a boolean ("true"). Apart from that, values are converted as appropriate; for instance, XSLT/XPath "numbers" are compatible with all numeric ABAP types (size permitting).

In Release 6.20, a more elegant mechanism was added for method calls with a single return value. Rather than spelling out the ABAP call pattern with its named parameters at each call, the pattern can be declared just once at the top level and method calls can then be performed in a functional syntax (with position parameters) *from XPath*, as shown in **Listing 6**.

An arbitrary namespace (for example, `xmlns:p="myFunctions"`) must be chosen for external functions defined this way, since the "null"

Listing 7: User-Defined XPath Functions

```
<!-- top-level function definition (recursive): -->
<sap:function name="p:factorial" >
  <sap:argument name="n"/>
  <sap:result define = "
    sap:if($n &lt;= 0, 0,
    sap:if($n = 1, 1,
    $n * p:factorial($n - 1) ) ) " />
</sap:function>

<!-- function call: -->
<xsl:value-of select="p:factorial(5)"/>
```

namespace is reserved for standard XPath functions. The rule that the first parameter of an instance method call is the object has been adopted as a quasi-standard in the "common Java binding mechanism" of Java-based XSLT processors.[15]

External calls (to Java) are quite often used with other XSLT implementations only because of a deficiency in the XSLT 1.0 specification: it does not provide a construct for *user-defined functions* to be called from XPath. Template rules are user-defined, but their purpose is to generate tree fragments and it is extremely awkward to compute, for example, a number or a string by a recursive template rule. The SAP XSLT implementation closes this gap by a top-level extension instruction, `sap:function`, illustrated in **Listing 7**.[16]

This instruction facilitates the definition of (mutually recursive) functions whose bodies are simply XPath expressions. Another XPath extension that is crucial to make this work is the *conditional expression* `sap:if(c, t, e)`, which evaluates to "t" if "c" is true, and to "e" otherwise. In Listing 7, `sap:if()` is used to terminate the recursion. Again, it is required that the user-defined functions lie in a namespace that is not null, the XSLT namespace, or the SAP-XSLT namespace. There may be several function definitions with the same name if they differ in the number of arguments.

Document-Centric Transformations

The major part of this article has been concerned with the use of XSLT for XML-to-XML mapping. (If this statement has you confused because the major part was supposed to be about XML-ABAP mappings, go back to Figures 2 and 3 — the mapping of an XML tree to an ABAP data structure works by means of an XSLT program mapping the XML tree to *another* XML tree in the "asXML" format, which is the canonical ABAP data representation. This result tree is *implicitly* constructed not as an XML tree, but as a

[15] In fact, since the common Java binding mechanism uses the namespace to identify Java classes, while the SAP mechanism specifies the ABAP methods in a construct ignored by other processors and doesn't care about the namespaces or local names of the functions defined, it is even possible to use the same XSLT program in an ABAP or Java context, provided that the external functions exist in both contexts.

[16] A similar construct will be included in XSLT 2.0 and will be implemented by SAP.

Recommended Reading

- ☑ Steven Holzner: *Inside XSLT*, New Riders 07/2001 (ISBN 0735711364)
- ☑ Michael Kay: *XSLT Programmer's Reference*, 2nd edition, Wrox Press 04/2001 (ISBN 1861005067)
- ☑ Michael Kay: *What kind of language is XSLT?*, 02/2001 (**www-106.ibm.com/developerworks/library/x-xslt/**)
- ☑ OASIS: *XML/XSL Cover Pages* (**www.oasis-open.org/cover/xsl.html**)
- ☑ W3C: *XSLT Specification*, 11/1999 (**www.w3.org/TR/xslt**)
- ☑ W3C: *XPath Specification*, 11/1999 (**www.w3.org/TR/xpath**)

data structure by the SAP XSLT processor. The direction "ABAP-to-XML" is symmetric.) Such applications are commonly called *data-centric*: the XML documents being transformed are typically not intended for direct viewing by a human user; rather, the goal of the transformations is to convert application data from one format to another. This contrasts with *document-centric* transformations, where the purpose is to bring information in some abstract XML format into (or closer to) a human-readable representation.

Of course, the SAP XSLT implementation can just as well be used for document-centric applications. Though the high-end formatting markup language XSL-FO is not implemented by the SAP kernel, any text-based output format can be generated with XSLT alone. The most important such format is certainly HTML. A typical practice is *device-dependent* layout: output is first produced in device-independent XML and then, depending on the display device, converted to a specific HTML dialect or some other format, like WML.

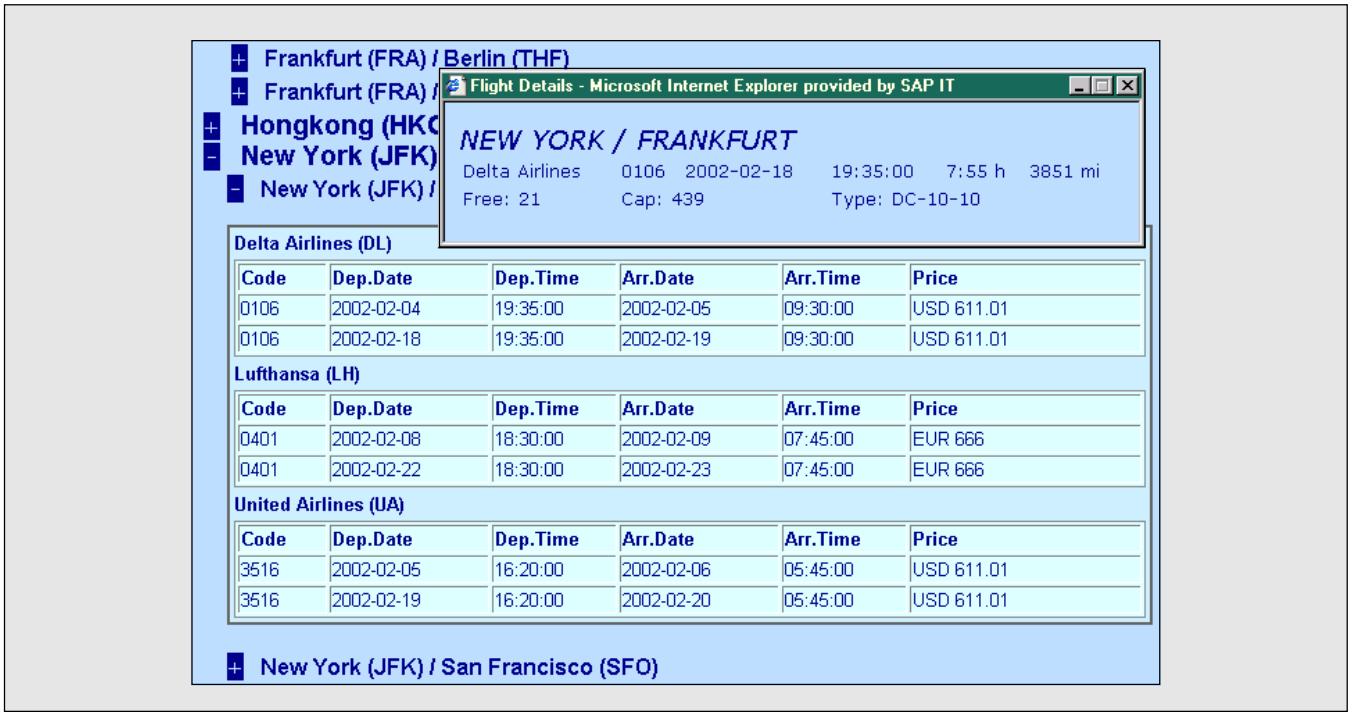
An example of an XML-to-HTML transformation is given in Listing 9 in the appendix (see page 55).

✓ *Business Server Pages and XSLT*

Business Server Pages (BSPs) are certainly the most important web technology provided by the SAP Web Application Server. BSPs and XSLT can be fruitfully combined. For instance, one way to implement the practice of device-dependent layout is to build a BSP application whose pages contain device-independent XML that is appropriately formatted by XSLT before sending it to the client. The invocation of XSLT can be done conveniently through a BSP extension (similar to a Tag Library). For a comprehensive discussion of the steps involved in creating BSP applications, see the article "A Developer's Guide to Creating Powerful and Flexible Web Applications with the New Web Application Builder" in the January/February 2002 issue of this publication.

Among other things, it illustrates that XSLT and *CSS* (Cascading Stylesheets) are not competing, but rather complementary technologies: XSLT is used to create the HTML structure, while CSS styles are used to control the appearance of certain HTML elements. Another complementary technology is Dynamic

Figure 7 HTML Output for an XFlight Document



HTML. In the last template of the example in Listing 9, additional information for each flight, retrieved from the database by an ABAP call, is inserted *statically* (at transformation time) into the HTML result, but "hidden" in a JavaScript call that will display this information *dynamically* (at display time) when the viewer clicks on a flight. The screenshot in **Figure 7** shows the result.

It follows that SAP XSLT can also be used to transform ABAP data structures more or less directly to a display format like HTML. In Listing 10 in the appendix (see page 58), we combine the programs of Listings 8 and 9 to generate the HTML output of Figure 7 via the intermediate "XFlight" structure. After more careful consideration, one might come to the conclusion that the literal English texts ("Dep.Date," "Price," etc. in Figure 7) appearing in the HTML output are an obstacle to internationalization. Therefore, SAP XSLT offers extension functions to access texts in the OTR (Online Text Repository), which makes it possible to efficiently isolate the transformation logic from language-specific content. This is analogous to the "text elements" of ABAP programs. Thus internationalization is yet another topic where the XSLT implementation builds on proven SAP server technology.

Conclusion

This article should have enabled you to answer the following main questions.

1. Why is XML important to ABAP-based development?
2. How can I bridge the gap between the ABAP world and the XML world?
3. What are the most important characteristics of XSLT in general and of the SAP XSLT implementation?

In summary, XML is an important technology because as a standardized, generic data model, it lays the basis for large-scale system integration, which is a current mega-trend in the IT industry. For ABAP developers, as for others, this presents opportunities

and challenges. In order to participate in open, XML-based communication standards, business software must be able to cope with a perpetually evolving set of data schemas. Hence, it would be futile to simply encode the data structures of an application interface in XML and expect it to be served by others. Likewise, it would be inadequate to shape an application's data model after some XML schema.

The SAP XSLT implementation offers a bridge between the communication data of XML and the processing data of ABAP. It is based on the definition of a canonical XML representation, asXML, of ABAP data structures. You can write XSLT programs that transform arbitrary XML to asXML (or vice versa), which, when applied to ABAP structures, actually write (or read) these structures. This solution is symmetric in that it supports arbitrary "schemas" on both sides and isolates the mapping between them in separate programs of a language that was designed for this purpose.

The implementation of the declarative, rule-based, high-level transformation language XSLT in the SAP Web Application Server reuses several important components of the proven SAP server architecture, including byte-code interpretation and program buffering. Integration into the workbench covers aspects like versioning and transport. Several extensions have been added to the language, most important the invocation of ABAP methods. A new instruction for calling XSLT has been added to ABAP, with special syntactic forms for data-centric transformations involving ABAP data structures. Classical "stylesheet" applications of XSLT (generating HTML, etc.) are supported just as well. Interoperability with the XML tools of SAP's "iXML" library is natural.

We feel that with these tools, XML handling in the ABAP environment becomes an enjoyable task. So, should you be confronted with an XML requirement sometime or other, do not despair: dig out this article, brush up on your `<xsl:instructions>`, and try it out for yourself!

Karsten Bohlmann received a diploma in computer science from the Technical University of Aachen, Germany, and a Ph.D. in computer science from the Technical University of Berlin. He joined a then-subsidiary of SAP in 1998. Since 2000, he has been a member of the Server Technology Group in SAP "basis" development. Karsten is the developer of the XSLT processor in the SAP kernel. He can be reached at karsten.bohlmann@sap.com.

Christian Fecht studied computer science at the University of Saarland in Saarbruecken, where he received his Ph.D. in 1997. He joined SAP in 1998, and since then has been working in the Business Programming Languages Group. He is responsible for the ABAP runtime environment, especially for the ABAP Objects garbage collector. Christian was also involved in the design of the Object Services, and recently his focus has been on transparent persistence for Java. He can be reached at christian.fecht@sap.com.

Christoph Wedler received a degree in computer science from the University of Erlangen, Germany, in 1993. He joined SAP in 1999 and is now a member of the Business Programming Languages Group. Christoph is responsible for the integration of XML into the ABAP language and different parts of the ABAP runtime environment. He can be reached at christoph.wedler@sap.com.

Appendix: Details of the asXML Format and Example Programs

This appendix accompanies the article "From XML to ABAP Data Structures and Back: Bridging the Gap with XSLT." It includes details of the ABAP serialization format asXML and three XSLT programs that define an ABAP-to-XML, XML-to-HTML, and ABAP-to-HTML transformation, respectively.

Details of the asXML Format

The Envelope

XML documents in the asXML format are easily recognizable by a fixed "envelope" that contains a list of symbolic names, under which there is the XML representation of an ABAP value. The root element `abap` in the asXML namespace `"http://www.sap.com/abapxml"` has an optional `version` attribute for future extensions. Its value is currently set to "1.0" by the ABAP-to-XML identity transformation.

Transformations may involve more than one ABAP variable. The asXML element `values` contains all the corresponding XML fragments as children. Each variable value is nested in an element with the respective binding name, as shown below:

```
<asx:abap version = "1.0" xmlns:asx = "http://www.sap.com/abapxml">
  <asx:values>
    <BINDING_NAME_1> ... </BINDING_NAME_1>
    ...
    <BINDING_NAME_n> ... </BINDING_NAME_n>
  </asx:values>
  <!--

... additional elements for objects and referenced data structures ...

-->
</asx:abap>
```

More elements may follow after the `values` element, but they are concerned with the representation of objects and data references, which is beyond the scope of this article.

Representation of Data Types

The representation of simple values conforms to the canonical representation of XML schema data types, as shown in the table on the next page.

A value of a *structure* type is represented as an element containing child elements that are named according to its components. Matching components are set by `MOVE-CORRESPONDING`. When transforming from XML to ABAP, additional sub-elements are ignored in XML input[17]; components without a corresponding XML element keep their value. In contrast with `MOVE-CORRESPONDING`, the "matching algorithm" is applied recursively.

An *internal table* is represented as an element containing child elements representing the lines of the table. When transforming ABAP to XML, the name of all child elements is the name of the line type if that type is a dictionary type, or "item" otherwise. When transforming from XML to ABAP, the name is irrelevant. Any kind of table (standard,

[17] However, sub-elements that have a name with a non-null URI component will lead to an exception, since they indicate a wrong input format.

ABAP Type	ABAP Example	XML Schema Type	XML Example	Comment
STRING	_Hello__	string	_Hello__	string of characters
C	_Hi	string	_Hi	
N	001234	string ([0-9]+)	001234	string of digits
I INT1, INT2	123-	int unsignedByte, short	-123	number
P	1.23-	decimal	-1.23	
F	-3.14E-2	double	-3.14E-2	
D	20020204	date	2002-02-04	ISO8601 date/time
T	201501	time	20:15:01	
XSTRING	456789AB	base64Binary	RweJqw==	Base64-encoded binary data
X	ABCDEF	base64Binary	q83v	
REF TO ...	not presented here			

sorted, hashed) is allowed; table lines for sorted tables do not have to be sorted as they will be sorted automatically when transforming from XML to ABAP.

Component and binding names could contain characters that are not legal for XML element names. These characters are mapped to a sequence of legal characters. For example, the illegal character `/` is mapped to the legal sequence `"_-"`. Other illegal characters are mapped to `"_--hh"`, where `hh` is the 2-character hexadecimal character code.

Example Programs

The XSLT programs listed here (**Listings 8-10**) are more complicated than the example shown back in Listing 4 (see page 38). It is necessary to understand the processing model of XSLT, which can be briefly summarized as follows:

- A `STACK` of node lists is processed to produce a `RESULT` tree from a `SOURCE` tree
- Start with `STACK = [ (ROOT(SOURCE)) ]`
- While `STACK` is not empty:
  - For each `NODE` in `TOP(STACK)`:
    - Find the (best) matching rule for `NODE`
    - Instantiate the rule body for `NODE`:
      - Append result nodes to `RESULT`
      - Instructions `xsl:apply-templates` and `xsl:for-each` push new node lists (in document order) on `STACK`
  - `POP(STACK)`

Listing 8 is an XSLT program that converts an ABAP flight list to the hypothetical XML standard XFlight, Listing 9 converts XFlight to HTML, and Listing 10 combines Listings 8 and 9 to generate HTML directly from ABAP. Note that closer inspection of Listing 8 reveals that this program is using some heavy machinery to achieve a goal that can be characterized as “grouping”: The flat flights document is transformed into the more structured connections document by forming groups of flights with common properties. The program is bulky because XSLT 1.0 isn’t good at grouping. XSLT 2.0 will have an `xsl:for-each-group` instruction that solves these problems very elegantly. The SAP XSLT processor will soon support this instruction.

Listing 8: XSLT Program That Converts an ABAP Flight List (in asXML) to XFlight

```
<xsl:transform  version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sap="http://www.sap.com/sapxsl"
>

<xsl:variable name="root" select="/"/>

<!-- ========================================================================
key for connections, using the "connection-key" {AIRPORTFR} '|' {AIRPORTTO}
======================================================================== -->
<xsl:key name="connkey"
         match="BAPISFLDAT"
         use="concat(AIRPORTFR,'|',AIRPORTTO)"/>

<!-- ========================================================================
  process flat flight list, produce list grouped by connections
======================================================================== -->
<xsl:template name="connections" match="FLIGHT_LIST">

<!-- collect all connections in $connections, sorted by connection-key -->
  <xsl:variable name="connections">
    <xsl:for-each select="*">
      <xsl:sort select="concat(AIRPORTFR,'|',AIRPORTTO)" data-type="text"/>
      <connect conn-key  = "{concat(AIRPORTFR,'|',AIRPORTTO)}"
               city-from = "{CITYFROM}"
               city-to   = "{CITYTO}" />
    </xsl:for-each>
  </xsl:variable>

<!-- output flights grouped by connections -->
  <connections>
    <xsl:for-each select=" sap:node-set($connections)/connect">
      <xsl:if test="not(@conn-key = preceding-sibling::connect[1]/@conn-key)">
        <xsl:call-template name="connection"/>
      </xsl:if>
    </xsl:for-each>
  </connections>

</xsl:template>

<!-- ========================================================================
  template for single connection
======================================================================== -->
<xsl:template name="connection">
  <xsl:variable name="conn-key" select="@conn-key"/>

  <connection from = "{substring-before($conn-key,'|')}"
              to   = "{substring-after($conn-key,'|')}">
    <xsl:copy-of select="@city-from | @city-to"/>

<!-- change source context back to original source, because key() is used -->
    <xsl:for-each select="$root">

<!-- collect flights for connection in $flights-by-carrier, sorted by AIRLINEID -->
      <xsl:variable name="flights" select="key('connkey',$conn-key)"/>
      <xsl:variable name="flights-by-carrier">
        <xsl:for-each select="$flights">
          <xsl:sort select="AIRLINEID"/>
```



```
          <flight carrier-code="{AIRLINEID}"  carrier-name="{AIRLINE}"
           code="{CONNECTID}">
            <departure>
              <date><xsl:value-of select="FLIGHTDATE"/></date>
              <time><xsl:value-of select="DEPTIME"/></time>
            </departure>
            <arrival>
              <date><xsl:value-of select="ARRDATE"/></date>
              <time><xsl:value-of select="ARRTIME"/></time>
            </arrival>
            <price>
              <amount><xsl:value-of select="PRICE"/></amount>
              <currency><xsl:value-of select="CURR"/></currency>
              <currency-iso><xsl:value-of select="CURR_ISO"/></currency-iso>
            </price>
          </flight>
        </xsl:for-each>
      </xsl:variable>

<!-- output flights for this connection, grouped by carrier -->
      <xsl:for-each select=" sap:node-set($flights-by-carrier)/flight">
        <xsl:variable name="carrier-code" select="string(@carrier-code)"/>
        <xsl:if test="not($carrier-code =
          preceding-sibling::flight[1]/@carrier-code)">
           <carrier code="{$carrier-code}" name="{@carrier-name}">
             <xsl:for-each select="
              . | following-sibling::flight[@carrier-code=$carrier-code]">
               <flight>
                 <xsl:copy-of select="@code | *"/>
               </flight>
             </xsl:for-each>
           </carrier>
        </xsl:if>
      </xsl:for-each>

    </xsl:for-each>

  </connection>

</xsl:template>

</xsl:transform>
```

Listing 9: XSLT Program That Converts XFlight to HTML

```
<xsl:transform  version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sap="http://www.sap.com/sapxsl"
>

<xsl:strip-space elements="*"/>
<xsl:output  method="html"  version="4.0"/>

<!-- ========================================================================
  process XML connection list (ordered by dep/arr airport), produce HTML
========================================================================= -->
```

(continued on next page)

(continued from previous page)

```
<xsl:template match="connections">
  <HTML>
    <HEAD>
      <STYLE type="text/css">
        BODY  {background-color:#bcdeff; color:#010188}
... more CSS styles ...
      </STYLE>
      <SCRIPT LANGUAGE="JavaScript">
        <xsl:text>
          function toggleDisplay(button, item) {
            if(button.value=='-') { item.style.display='none'; button.value='+'; }
            else                  { item.style.display=''; button.value='-'; }
          }
          function infoWindow(wid, cfr, cto, car, cod, dat, tim, dur, dis, pty,
                              fre, cap) {
... JavaScript to pop up information window ...
          }
        </xsl:text>
      </SCRIPT>
    </HEAD>
    <BODY>
      <TITLE>Flight Connections</TITLE>
<!-- for each departure airport, process first connection (beginning of group) -->
      <xsl:apply-templates
        select="connection[not(@from=preceding-sibling::connection/@from)]"
        mode="first"/>
    </BODY>
  </HTML>
</xsl:template>

<!-- ============================================================================
  template for group of connections with identical departure airport
============================================================================= -->
<xsl:template match="connection" mode="first">
  <xsl:variable name="from" select="@from"/>
<!-- title for this group -->
  <DIV class="airport">
    <INPUT type="button" value="-" style="cursor:hand"
     onclick="toggleDisplay(this, {$from});"/>  
    <xsl:value-of select="@city-from"/> (<xsl:value-of select="$from"/>)
  </DIV>
<!-- table of all connections in this group (including the first connection) -->
  <TABLE id="{$from}" width="100%">
    <xsl:apply-templates select=". | following-sibling::connection[@from=$from]"/>
  </TABLE>
</xsl:template>

<!-- ============================================================================
  template for single connection
============================================================================= -->
<xsl:template match="connection">
  <xsl:variable name="id" select="concat(@from,'_',@to)"/>
  <TR><TD> </TD><TD>
<!-- title for this connection -->
    <DIV class="conn">
<!-- open/close button for this connection -->
      <INPUT type="button" value="+" style="cursor:hand"
       onclick="toggleDisplay(this, {$id});"/>  
      <xsl:value-of select="@city-from"/> (<xsl:value-of select="@from"/>) /
```



```
      <xsl:value-of select="@city-to"/> (<xsl:value-of select="@to"/>)
    </DIV>
<!-- table of carriers for this connection (initially closed) -->
    <DIV id="{$id}" style="display:none">
      <BR/>
      <TABLE border="0" class="conn" width="100%">
        <xsl:apply-templates/>
      </TABLE>
      <BR/>
    </DIV>
  </TD></TR>
</xsl:template>

<!-- ========================================================================================
  template for carrier in connection
========================================================================== -->
<xsl:template match="carrier">
<!-- title for this carrier -->
  <TR><TD>
    <DIV class="carrier">
      <xsl:value-of select="@name"/> (<xsl:value-of select="@code"/>)
    </DIV>
  </TD></TR>
<!-- table of flights for this carrier -->
  <TR><TD>
    <TABLE class="flight" border="1" width="100%">
      <TR class="head">
        <TD width="10%">Code</TD>
        <TD width="18%">Dep.Date</TD>
        <TD width="14%">Dep.Time</TD>
        <TD width="18%">Arr.Date</TD>
        <TD width="14%">Arr.Time</TD>
        <TD width="26%">Price</TD>
      </TR>
      <xsl:apply-templates/>
    </TABLE>
  </TD></TR>
</xsl:template>

<!-- ========================================================================================
  template for flight by carrier in connection
========================================================================== -->
<xsl:template match="flight">
  <xsl:variable name="carr" select=".."/>
  <xsl:variable name="conn" select="$carr/.."/>
  <TR class="flight">
    <sap:call-external  class  = "CL_SXSLTDEMO_FLIGHT_INFO"
                        method = "FLIGHT_GETDETAIL">
      <sap:callvalue    param  = "AIRLINEID"    select="string($carr/@code)"/>
      <sap:callvalue    param  = "CONNECTIONID" select="string(@code)"/>
      <sap:callvalue    param  = "FLIGHTDATE"   select="string(departure/date)"/>
      <sap:callvariable param  = "FLIGHTTIME"   name="duration"/>
      <sap:callvariable param  = "DISTANCE"     name="distance" type="number"/>
      <sap:callvariable param  = "UNIT"         name="unit"/>
      <sap:callvariable param  = "PLANETYPE"    name="planetype"/>
      <sap:callvariable param  = "ECONOMAX"     name="emax"/>
      <sap:callvariable param  = "ECONOFREE"    name="efree"/>
      <sap:callvariable param  = "BUSINMAX"     name="bmax"/>
      <sap:callvariable param  = "BUSINFREE"    name="bfree"/>
```

(continued on next page)

(continued from previous page)

```
      <sap:callvariable param  = "FIRSTMAX"     name="fmax"/>
      <sap:callvariable param  = "FIRSTFREE"    name="ffree"/>
    </sap:call-external>
    <xsl:attribute name="onclick">
      infoWindow(
<!-- insert first half of JavaScript parameters: basic flight information -->
      <xsl:value-of select='concat("&apos;",
        translate(concat($conn/@from,$conn/@to,departure/date)," -:","___"),
        "&apos;,&apos;", $conn/@city-from, "&apos;,", "&apos;", $conn/@city-to,
        "&apos;,&apos;", $carr/@name, "&apos;,", "&apos;", @code,
        "&apos;,&apos;", departure/date, "&apos;,", "&apos;", departure/time,
        "&apos;,")'/>
<!-- insert second half of JavaScript parameters: additional flight information -->
        <xsl:value-of select='concat(
          "&apos;", $duration, " h", "&apos;,",
          "&apos;", $distance, " ", $unit, "&apos;,",
          "&apos;", $planetype, "&apos;,",
          "&apos;", $efree + $bfree + $ffree, "&apos;,",
          "&apos;", $emax + $bmax + $fmax, "&apos;" )'/>
      );
    </xsl:attribute>
    <TD><xsl:value-of select="@code"/></TD>
    <TD><xsl:value-of select="departure/date"/></TD>
    <TD><xsl:value-of select="departure/time"/></TD>
    <TD><xsl:value-of select="arrival/date"/></TD>
    <TD><xsl:value-of select="arrival/time"/></TD>
    <TD><xsl:value-of select=" concat(price/currency-iso, ' ',
                               number(price/amount))"/></TD>
  </TR>

</xsl:template>

</xsl:transform>
```

Listing 10: XSLT Program That Converts an ABAP Flight List to HTML

```
<xsl:transform  version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sap="http://www.sap.com/sapxsl"
>

<xsl:include sap:name="SXSLTDEMO_FLIGHTS_CONNECTIONS"/>
<xsl:include sap:name="SXSLTDEMO_CONNECTIONS_HTML"/>

<xsl:template match="FLIGHT_LIST">
<!-- create connection list from flight list -->
  <xsl:variable name="connection-list">
    <xsl:call-template name="connections"/>
  </xsl:variable>
<!-- produce HTML document for connection list -->
  <xsl:apply-templates select=" sap:node-set($connection-list)"/>
</xsl:template>

</xsl:transform>
```

