# Why XSL is not suitable for Semantic Translation

Gabor Karsai
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235

Recently, XML [XML] has been extended with a capability, called XSLT, [XSL] for translating XML data compliant with one DTD into XML compliant with another DTD. This approach has been hailed as the general solution for data transformation in database systems that can solve *any and all* data integration problems by providing a simple way for transforming data. In this note I will briefly review the XSLT approach and contrast it with another approach developed for tool integration.

## XML and XSLT

XML [XML], a descendant of SGML and HTML, is an extensible markup language that supports structured data. As opposed to ASCII (which supports un-structured data), XML documents have a well-defined structure, which is (usually but not necessarily) specified in a form called DTD (Document Type Definition). When data is expressed in XML, the data is usually accompanied by a corresponding DTD that captures the structuring rules. One can think about XML documents as data that comply with a certain syntax that is defined by a DTD. When an XML document is processed, the DTD is used to configure a parser engine, which then reads the main XML body. The parser can verify that the data complies with the structuring rules, and, for example, can build a parse-tree from the data that follows the syntactical rules specified in the DTD.

XSLT [XSL] is a capability that can transform XML documents compliant with one DTD into documents compliant with another DTD. XSL transformations are defined as an XML document (that is compliant with the XSLT DTD). The transformation engine reads these transformation rules and configures itself to act as a rewriting tool. This process is illustrated on the diagram below.

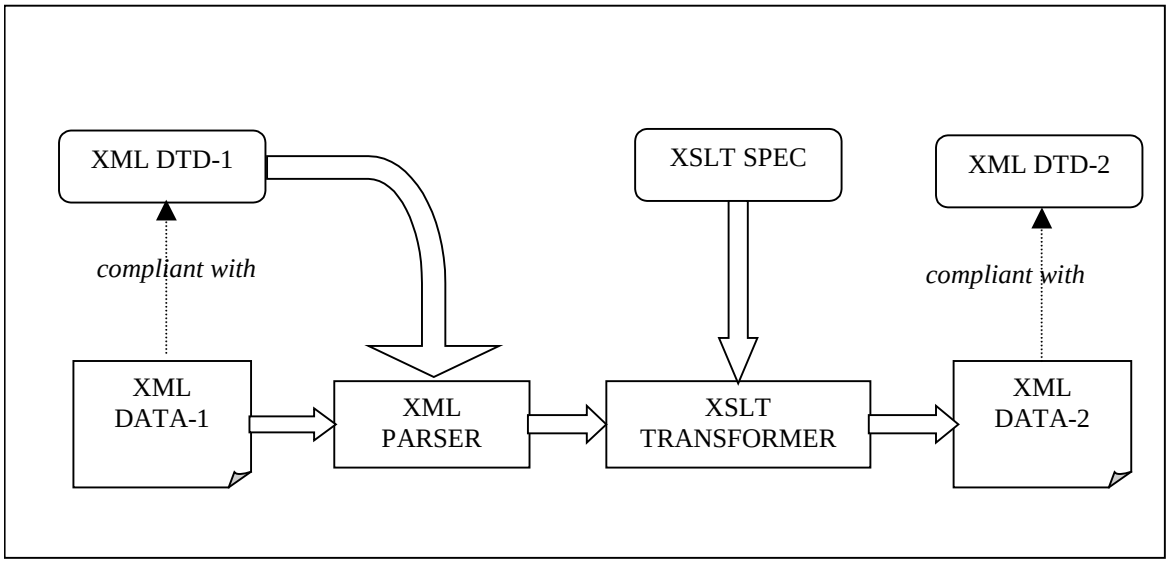


**Figure 1: XML translation process using XSLT**

The translation process is tied to the low-level data model available in XML. All XML documents are trees with attributed nodes. Parent-children relations are qualified by names, and children nodes can be reached by traversing from the parent node using selectors that specify what kind of children nodes are to be reached. The XSLT approach lets one define a translation process in terms of tree-rewriting rules.

A rewriting rule has two main components: (1) a *pattern* that has to match a sub-tree of the XML document, and (2) an *output spec* that specifies what the input (sub-)tree has to be translated into. The translation engine works as follows.

1. It reads the first tree in the XML document and looks for a matching rewriting rule.
2. If it finds a rule whose pattern matches the tree, it processes its output spec. The output spec may contain simple output instructions (i.e. text that has to be sent to the output), and/or other control commands that instruct the engine to recursively process all or one of the sub-trees of the node.

There are sophisticated capabilities in the XSLT language to specify

- patterns that match nodes or entire sub-trees,
- access to nodes in the tree from a given point,
- access to attributes of nodes, and
- building up the output XML tree structure, in general.

To summarize, XSLT provides a mechanism for describing transformations on XML data that is compliant with a specific DTD. The transformations are described in terms of rewriting rules, that are matched against the input, and specify how the output should be constructed from all or parts of the matching input.

## Tool Integration Framework

Recent work [TIF] on solving the tool integration problem has focused on using an architectural approach combined with semantic translators. The tool integration problem is defined as providing an infrastructure for interchanging data between any two engineering tools (CAD packages, databases, analysis tools, etc.) such that only semantically correct data is received by any tool, while keeping integration costs low. The solution was partly based on an architecture consisting of an Integrated Model Server (IMS) and Tool Adaptors (TA) linked through a CORBA-based protocol.

The approach incorporates semantic data modeling. For each tool a detailed data model is prepared that captures the static semantics of the tool's data in the form of attributed *models* (containers), *entities* and *relations*, together with non-structural *constraints* expressed in UML's OCL [UML]. From these data models an integrated data model is created that has a schema "rich enough" to capture data expressed in any of the other data models. The IMS is responsible for providing semantic translation services between the tool's data models and the integrated data model. (Data in the integrated data model form is also archived in a persistent storage facility.) The TA-s are used to convert physical tool data into the protocol form (which is canonical) and vice versa. Each protocol data item is tagged with a type-code that identifies what the data item means in the form of the tool's data model.

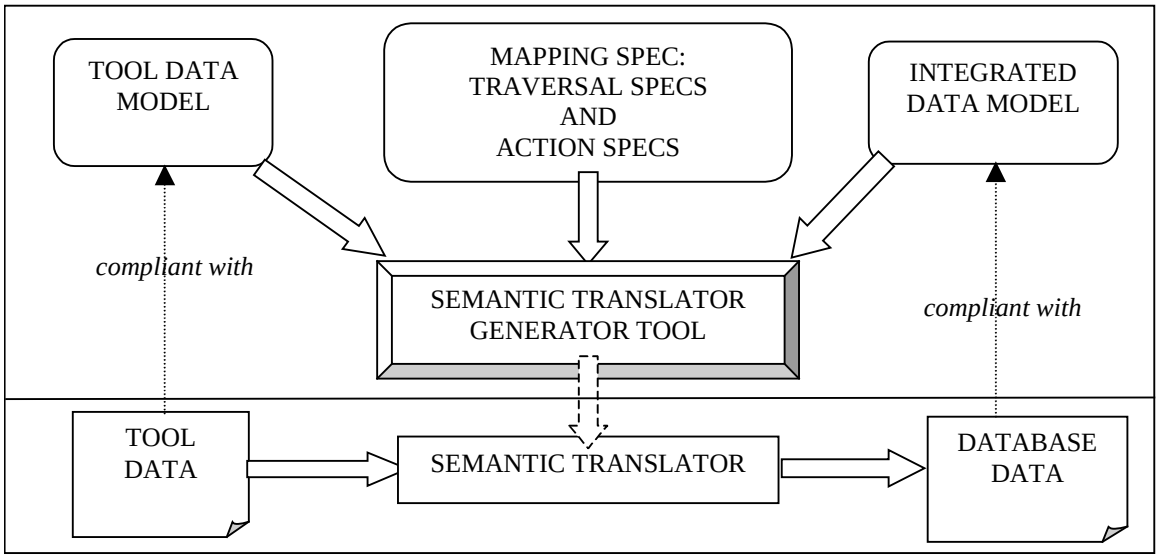


**Figure 2: Generating and using a semantic translator**

The IMS contains *semantic translator* components that are responsible for performing transformation on the data and enforcing the static semantic constraints. In the TIF approach these translators are generated from high-level specifications as follows. A translator has two "sides": a tool side, and a database side. The tool side is compliant with the tool's data model, and the database side is compliant with the integrated data model. The task of the translator is to

implement the mapping between the two data models and then to enforce the constraints on the output. This mapping process is expressed in a language that has two components:

1. Traversal specs to describe how the input data-structure should be traversed
2. Actions specs that specify what actions are to be taken when visiting nodes of a specific type.

The traversal specs are described in a structured language while the actions are expressed in the form of embedded C++ code. The data models and the mapping specifications are processed by a generator tool that creates efficient C++ code that "fits" into the framework. The translation and the translation generation process is illustrated on Figure 2. The described approach has been used in integrating a number of tools and databases. The translation approach is inherently procedural (although some algorithmic details are hidden behind a high-level language describing traversal sequences), and thus highly efficient. The data models have, in addition to the data schema, constraints that are Boolean expressions. After translation, these expressions are evaluated in the context of the result of the translation. If a constraint evaluates to FALSE, that implies a violation of the static semantics of the data model, and an error is signaled.

## Comparison

It is natural question to ask how the approaches used in XSLT and TIF compare. Obviously, the two approaches are similar in nature, both in terms of goals and implementations. Below, I will summarize the major differences and show how the two approaches are suitable for different classes of transformation problems.

1. XSLT follows the simple tree-structured data model of XML, while TIF has a sophisticated, yet clean, data model suitable for representing arbitrary graphs (with attributed nodes and edges). While it is possible to express graphs in XML (through ID-s and/or links), these are outside of the underlying data model (i.e. the tree), and they are non-trivial to process (i.e. traverse) using the XSLT capabilities. The main problem is related to efficiency: following a link that uses an ID triggers a search for an object in the tree with the matching ID.
2. The XSLT transformations are always coupled to a pattern matching process (i.e. graph search), while in TIF the programmer has explicit control over the graph traversals. While control over traversals is always possible in XSLT, the choices are usually limited to the tree-structure, thus arbitrary traversals are difficult to write. Pattern matching vs. explicit traversals also have performance implications: while explicit traversals are admittedly lower level than patterns, they do not cause performance penalties.
3. XSLT does not have any high-level constraints on the data it processes (except the structural constraints imposed by the DTD). In TIF constraints are applied in two forms: (1) All data objects are typed and the run-time system — together with the C++ compiler — enforces type compatibility between the translator algorithm and the data it processes and generates. (2) Data models include high-level constraint specs as well that are evaluated after the translation has finished. These constraints capture and enforce the static semantics of the data model at hand, and thus translators cannot produce output that would be semantically incorrect (i.e. non-compliant with the constraints specified).
4. XSLT has interface to XML data only: XSLT rewriting rules directly operate on XML data. The TIF approach has interface to data in any form (including data accessible only through an API), via the tool adapters, by design. The TA-s are not responsible for semantic translation, thus they are very easy to develop. Naturally, TA-s can be developed for XML as well (to convert tool data into XML).
5. The XSLT rewriting approach makes it very hard to attach arbitrary computations to the translation process. In TIF arbitrary computations can be attached to any traversal actions.
6. In TIF, a very useful feature is to dynamically build, maintain, and "pass-around" a data-structure —a context— during the translation process, similar to the attributes in attributed grammars in compilers [CMP]. XSLT lacks similar facilities.
7. XSLT is difficult to use when the output structure is dependent on the input (attribute) data. For instance, in a TIF application, data in a flat data model was to be mapped into a hierarchical data model (and vice versa). The hierarchical structure had to be generated based on attribute values on the input objects. It required multiple passes over the input structure, and establishing the hierarchy in the second pass. To do this in XML one has to develop multiple XSLT "stages" that are pipelined: a rather complicated procedure.
8. XSLT uses an "interpreted" approach: it reads the DTD-s and the XSLT specs and uses them to configure the parser and the translation engine. In TIF we have used —automatically generated — efficient and compact C++ code accomplish similar results, with considerably better performance.
9. We have to recognize that XSLT provides a simple and effective method for translating data from one data model into the other *if the mapping is easy to express using XSLT's mechanisms*. Presumably, a very large

portion of useful and necessary data transformations falls into this category, and if the translation performance is not an issue, XSLT is a suitable choice for a technology.

## Conclusions

In this note I have reviewed and compared two approaches for data translation. One of them is based on and is a component of the XML technology, currently under development. The other approach used an architectural framework with some components generated from high-level specifications. The first approach, XSLT, seems suitable for translating tree-structured XML documents, through a pattern-matching/rewriting process. For transformations where this approach is suitable (both in terms of the sophistication of the transformation and performance), this is a good and simple way to do it. The second approach is based on a universal framework that can process data in any form, and is suitable where a complex mapping is necessary and/or the data semantics has to be enforced. I would recommend the use of the first approach for easy tasks, but would prefer the second one for complex, more robust solutions. While XSLT offers quick and simple solutions for simple translations (which are very easy to write once the data is available in XML form), the TIF approach can be used whenever the full power of a programming language is needed to facilitate the translation.

## References


[CMP] Aho,A., Sethi,R., Ullmann: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[TIF] Karsai G., Gray J.: "Component Generation Technology for Semantic Tool Integration", Proceedings of the IEEE Aerospace 2000, CD-ROM Reference 10.0303, Big Sky, MT, March, 2000.

[UML] Martin Fowler, *UML Distilled*, 2nd ed., Addison-Wesley, 1999.

[XML] Elliotte Rusty Harold: XML Bible, IDG Books Worldwide, 1999.

[XSL] http://metalab.unc.edu/xml/books/bible/updates/14.html