



# XML mapping technology: Making connections in an XML-centric world



M. Roth  
M. A. Hernandez  
P. Coulthard  
L. Yan  
L. Popa  
H. C.-T. Ho  
C. C. Salter

Extensible Markup Language (XML) has grown rapidly over the last decade to become the *de facto* standard for heterogeneous data exchange. Its popularity is due in large part to the ease with which diverse kinds of information can be represented as a result of the self-describing nature and extensibility of XML itself. The ease and speed with which information can be represented does not extend, however, to exchanging such information between autonomous sources. In the absence of controlling standards, such sources will typically choose differing XML representations for the same concept, and the actual exchange of information between them requires that the representation produced by one source be transformed into a representation understood by the other. Creating this information exchange “glue” is a tedious and error-prone process, whether expressed as Extensible Stylesheet Language Transformation (XSLT), XQuery, Java™, Structured Query Language (SQL), or some other format. In this paper, we present an extensible XML mapping architecture that elevates XML mapping technology to a fundamental integration component that promotes code generation, mapping reuse, and mapping as metadata.

## INTRODUCTION

The advent of the Internet and standard transfer protocols have made it easier for enterprises to programmatically exchange information, and Extensible Markup Language (XML) has grown rapidly over the last decade to become the *de facto* standard for this data exchange. Indeed, investments in architectures such as Enterprise Information Integration (EII)<sup>1</sup> and service-oriented architecture (SOA)<sup>2</sup> demonstrate that integration initiatives occupy a significant portion of information technology spending. While XML and related technologies have made heterogeneous data exchange

possible, actual information exchange is not yet automatic—or even easy. A primary reason is that autonomous sources of information, in the absence of controlling standards, typically choose differing XML representations for the same concept, and information exchange between them requires that the representation produced by one source be

©Copyright 2006 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/06/\$5.00 © 2006 IBM

transformed into a representation understood by the other.

Structured Query Language/XML (SQL/XML),<sup>3</sup> Extensible Stylesheet Language Transformation (XSLT),<sup>4</sup> and XQuery<sup>5</sup> are transformation languages that can be used to specify how to transform XML from one format to another, and several XQuery and XSLT transformation engines exist that can process such transformations.<sup>6,7</sup> Although such languages are quite powerful, it can often be very difficult to express the transformation by hand in these languages. For example, consider a simple transformation that combines a customer's personal, account, and bank transaction information into a single XML document for use by a bank teller application. *Figure 1* illustrates the transformation and the XQuery that expresses it.

Although the visual relationships shown at the top of *Figure 1* are intuitive, the XQuery statements demonstrate that expressing even this simple transformation is quite verbose, and to do so by hand is tedious and error-prone. The three source documents may contain multiple instances of personal, account, and transaction information, and the complexity of the XQuery specification is to ensure this information is properly joined and grouped to associate clients with their accounts and accounts with their transactions.

Mapping technology helps to automate this process. A visual mapping tool is used at design time to specify a high-level, language-neutral description (a *mapping*) of how to turn thing A (e.g., descriptions of the personal, account, and transaction information in *Figure 1*) into thing B (e.g., a description of the bank teller's view in *Figure 1*). The mapping tool translates the mapping into executable code, which can then be deployed to a runtime engine that can repeatedly perform the transformation over multiple input data streams.

Although the example in *Figure 1* used instance data to illustrate the transformation and indeed, at runtime the transformation occurs over large volumes of instance data, a mapping tool usually works with descriptions of instance data. Implicit in this statement is the assumption that instance data adheres to a structure that can be described in a machine-readable format. (Unstructured mapping, such as the derivation of structured information

from a text document, spreadsheet, and so forth, is an area rich with requirements and technology, but is outside the scope of this paper.)

Since the advent of XML, at least one of thing A or thing B is usually an XML schema, while the other can be described in a number of different ways, such as another XML schema or Document Type Definition (DTD), a relational schema, a COBOL (common business-oriented language) copybook, or one of many other legacy data formats. Other possible variations include transforming XML documents by means of XQuery or XSLT specifications or generating an XML document from relational tables by means of SQL/XML statements. Another important case is to transform data from XML to another format. Legacy systems running in a mainframe environment must frequently adapt to accept XML as input, such as a mainframe-based order system written in COBOL adapting to process orders submitted through a Web application. Note also that as information flows through an enterprise, it may be necessary to connect a series of mapping steps together, and as a result, thing A and thing B may be part of a larger topology of mapping steps.

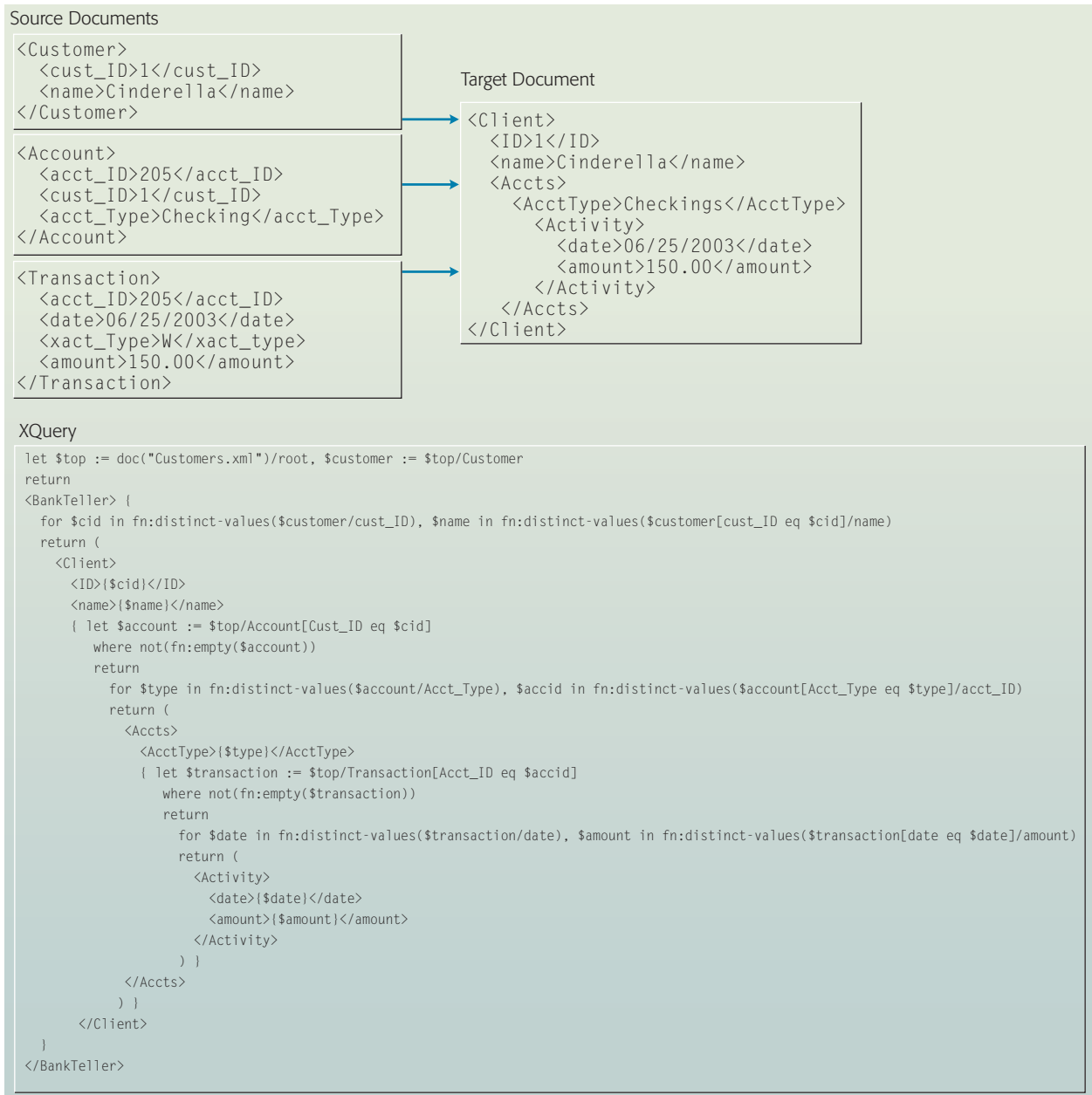
In this paper, we discuss the current state of XML mapping technology and its requirements, present an extensible framework and architecture that address those requirements, and discuss the challenges that remain. We describe a set of motivating examples drawn from different contexts—EII and SOA—and include an overview of requirements that can be derived from these and other scenarios for software tools that enable and automate mapping and transformation. We then describe an extensible architecture for a mapping tool to address these requirements, followed by a description of a set of research challenges that still remain.

## MOTIVATING EXAMPLES

There are many places to look for motivating examples for XML mapping technology. In this section we focus on two important areas that have arisen from the need to integrate existing assets: EII—driven by the need to bring large volumes of disparate information together, and SOA—driven by the need to orchestrate applications into integrated business processes.

### Enterprise Information Integration

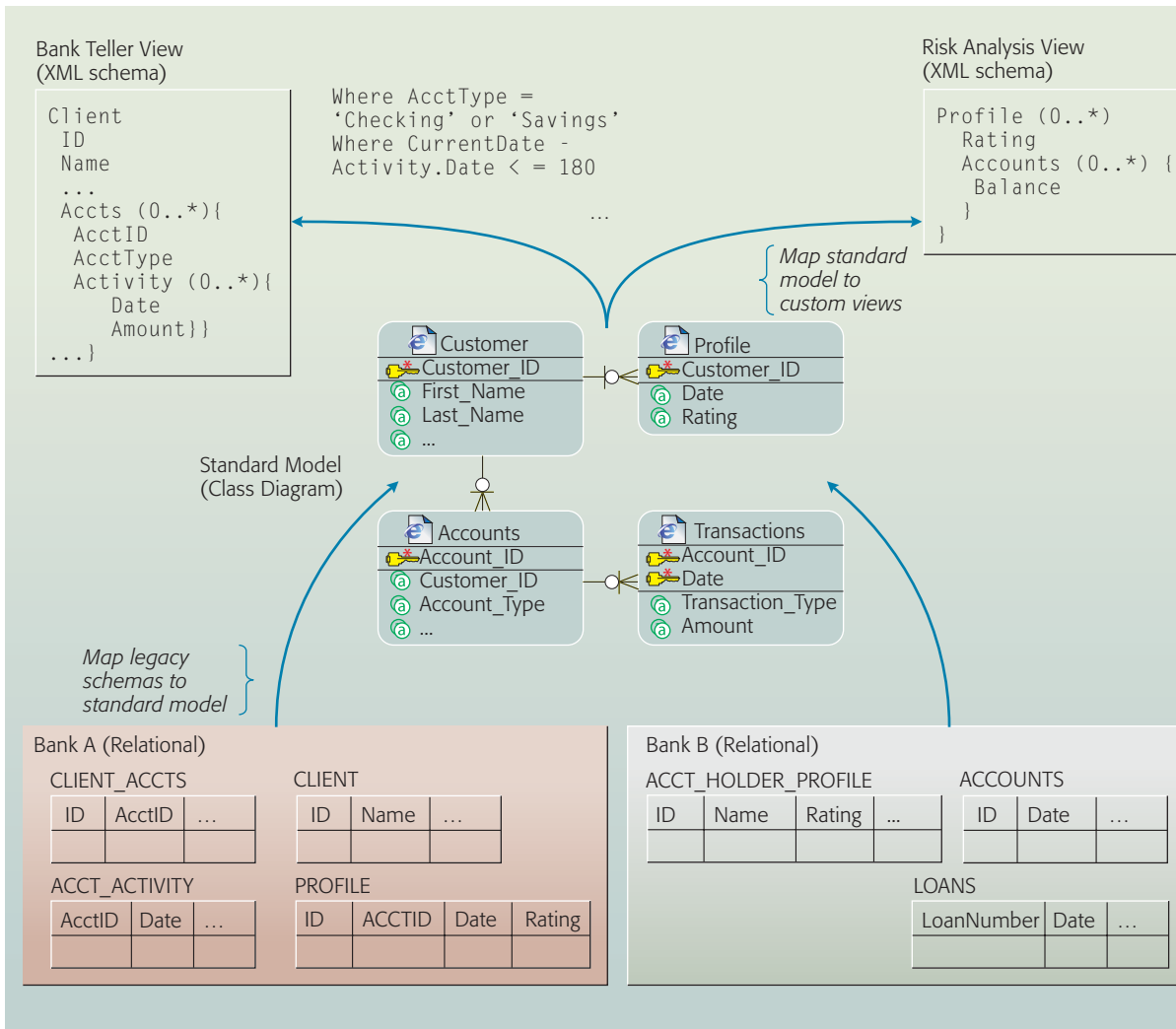
Information captured, processed, and produced by enterprise applications is growing at an enormous



**Figure 1**  
A simple transformation expressed in XQuery

rate, and the need to connect and unify such information is growing just as rapidly. EII provides a unified view of related information from disparate systems in real time without copying or moving the data to a central repository or warehouse. Many software vendors provide EII products, including IBM,<sup>8</sup> MetaMatrix,<sup>9</sup> and Sify.<sup>10</sup> The runtime engine for EII is typically a database engine or its equivalent, and the transformation language is SQL.

Let us consider a typical (if simplified) scenario for EII. A banking institution offers multiple financial services, including, for example, checking and savings accounts and loans. These services are provided by multiple overlapping departments that represent acquisitions of other institutions. Each department has autonomously created its own representation for customers, its own financial arrangements with those customers, and its own



**Figure 2**  
An Enterprise Information Integration (EII) example

vocabulary of terms—and stores that information in a set of relational tables. For example, Bank A tracks all arrangements (loans, checking accounts, and savings accounts) in a common set of tables, while Bank B tracks each type of arrangement separately.

The data architect for an institution that acquired both banks wants to consolidate the different representations of loan and account information into a unified model with a common structure and a common set of terms. This is needed to foster a global understanding of the information captured and processed by the bank. As a secondary step, the architect would like to provide tailored views of that information which can be embedded in different applications destined for different user roles

throughout the institution. These views are often restricted to a subset of the information (determined by the user role and authorization), and they require restructuring, renaming, and reformatting of the data.

*Figure 2* illustrates a potential solution for this scenario. The data architect can define a standard model and define maps from the existing departments to the standard model that specify how to rename and restructure the department information in terms of the standard model. In this scenario, the data architect has represented the standard model in terms of a class diagram expressed in Unified Modeling Language\*\* (UML\*\*).<sup>11</sup> He has then defined maps from the Bank A relational schema to

the standard model and from the Bank B relational schema to the standard model. Additionally, a map has been defined from the standard UML model to the customized views needed by two different Web applications. Because XML is a more suitable data format for Web applications, these views are defined by XML schemas. The first schema represents information accessible by bank tellers, who are authorized to see the last six months of checking and savings account activity by a particular customer, but who are restricted from seeing information about the customer's loan information and credit risk profile. The second schema represents information accessible by risk analysts. They are allowed to see a detailed view of the account balance and credit risk rating for all customers, but personal identifying characteristics are removed from the data.

This scenario illustrates a rich set of requirements for XML mapping technology. Notice that the source and targets of the mappings are represented by three different formats: relational, UML, and XML Schema. In addition, data must be joined (e.g., the CLIENT, CLIENT\_ACCTS, ACCT\_ACTIVITY, and PROFILE on the Bank A schema), and unioned (the Bank A and Bank B schemas). Data is aggregated (a credit rating per customer per time period to a single rating per customer) and conditionally transformed to the final XML representation (bank tellers can see only checking and savings account activity for the past six months). The scenario also illustrates a need to combine maps two ways: in parallel, to combine the Bank A and Bank B schemas to form the standard model; and serially, to compose the final XML schema as a single transformation rather than as a two-step process—from existing relational schemas to the standard model to the customized schema.

Finally, notice the difference in terminology used by each of the data models. A mapping tool that requires a user to manually match elements by hand would be at best tedious (e.g., BankA.PROFILE.Rating → Standard.Profile.Rating), and at worst impossible if there is no obvious semantic information that indicates two differently named elements refer to the same concept. For example, suppose the standard model includes “Alternate Address” for clients, and the BankA.CLIENTS table contains a column called ADDRESS2. Is ADDRESS2 the second line

of the client's primary address, or is it an alternate address?

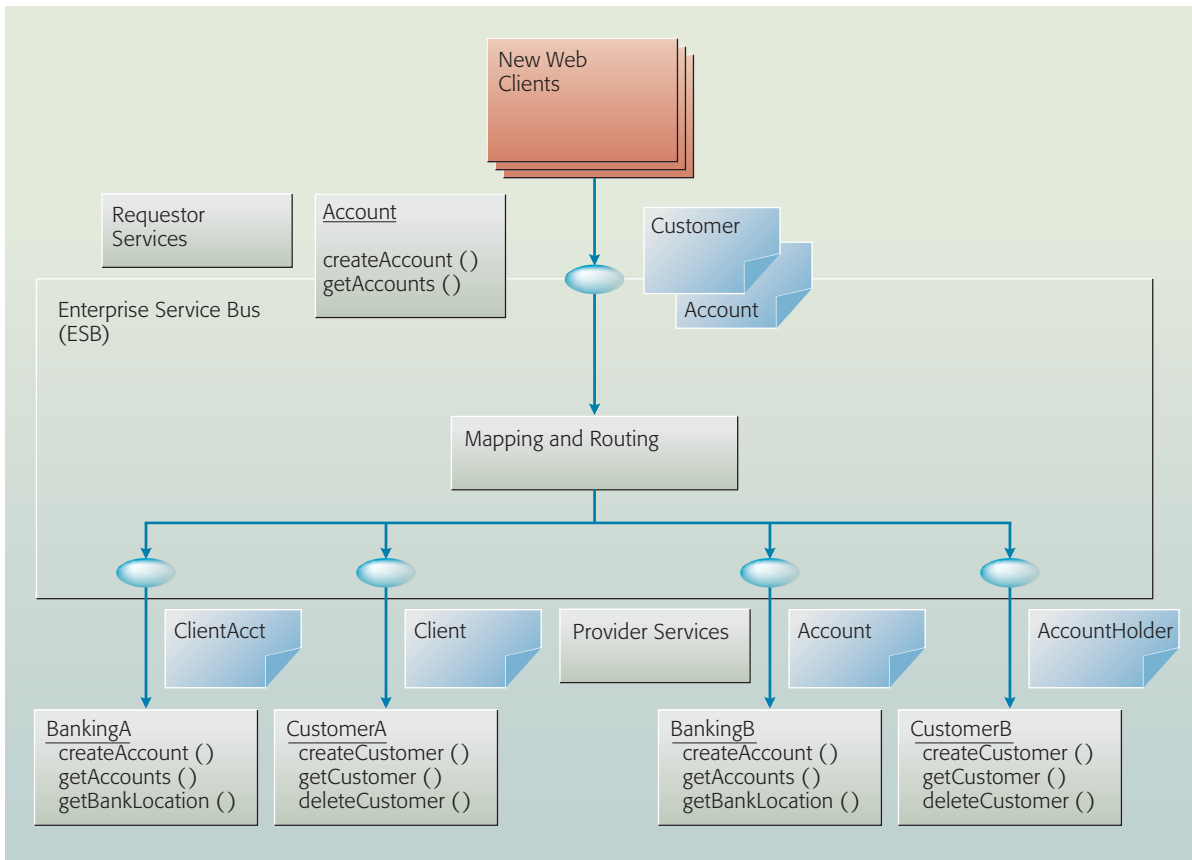
### Service-oriented architecture

While EII is focused on real-time information integration, SOA is focused on integrating applications at the interface layer and integrating the data exchanged between them. SOA and its predecessor, Enterprise Application Integration (EAI), have become strategically important for enterprises to increase information technology efficiency by reusing and integrating existing assets and to improve quality and customer satisfaction through automation.<sup>2</sup>

*Figure 3* illustrates an SOA implementation of the EII banking institution scenario. The Enterprise Service Bus (ESB) publishes a requestor service, named Account, with operations `createAccount()` and `getAccount()` to be invoked by the new Web application. This requestor service is supported on the ESB by routing to the appropriate customer and banking provider services registered to the ESB by the bank's subsidiaries. In such a model, the customer service offers life-cycle operations, like `create`, `delete`, and `update`, and the banking service offers typical banking operations, like `createAccount`, `getAccounts`, and `getBankLocation`.

Message mapping (that is, the service parameters) is key to enabling transparency, and designing maps between services represents a major focus for SOA architects. XML schemas for messages can grow large and use complex constructs, such as attribute groups, redefines, extensions, and restrictions. Mapping expressiveness in SOA is reminiscent of the EII scenario, including joins, conditions, scalar functions (e.g., concatenate first and last name to produce `full_name`), and set-based functions (e.g., compute the sum in a repeating list of numeric values or sort the result by a particular attribute).

Both the EII and SOA scenarios underscore the importance of XML mapping technology and draw out a number of algorithmic requirements. It is also important to note the skill level of the users in both of these scenarios. For instance, data architects and integration specialists, while technically knowledgeable, are not necessarily programmers. As a result, ease-of-use features, such as highly visual editors, visual differentiation for various types of



**Figure 3**  
A banking scenario implemented on an SOA architecture

maps (e.g., joins compared with unions), live output samples, tree views, graphical views, and source views (for the more highly skilled)—are critically important for any set of mapping tools used for this community of users. Further, rich problem determination tools are also important, including visual debugging, real-time validation with error correlation, and runtime trace support.

### STATE OF THE ART

Many tools are specialized to address a specific business problem or are targeted at specific runtime execution environments and thus only produce executable code in a single language (such as XSLT). Other mapping tools are loosely coupled from the runtime execution language and are capable of generating to multiple executable transformation languages. Mapping tools generally follow one of two visual metaphors, either highlighting the relationships between a source and target (*relationship based*) or highlighting the functional steps

required to transform a source to a target (*function based*).

### Relationship-based mapping tools

Relationship-based mapping tools typically display source data structures on the left side and target data structures on the right side. Relationships between fields and sets of fields in the source and target data structures are visualized by connecting lines (though there are some exceptions). In many tools, relationships can be annotated with functions, conditions, and other transformation information.

The relationships and the semantics implied by the source and target descriptions are captured in a data structure, which is typically saved in a proprietary format and used to generate transformation code. Some tools maintain an intermediate representation and generate the transformation logic in multiple executable languages, such as XSLT, XQuery, and Java\*\* code. The intermediate representation in-



roduces the difficulty of keeping the mapping specification and the generated code synchronized. An alternative approach, such as that used by Progress Software Stylus Studio\*\*, is to save the mapping specification directly as generated code. This means that the generated transformation code can be reinterpreted and visualized when loaded back into the tool.

A relationship-based approach offers a nice abstraction, particularly for less technical users. However, a relationship-based tool is generally not Turing-machine equivalent (any computable function can be expressed and computed by the software), and therefore cannot be used to express every possible transformation. Relationship-based tools handle this by providing some kind of user exit that allows a user to write directly in the target language, typically with an integrated language editor. Furthermore, because relationship-based tools operate at a higher level of abstraction than executable code (e.g., one line may correspond to several lines of code or several function invocations), debugging and testing features typically operate directly on the generated transform rather than the mapping specification itself.

### **Function-based mapping tools**

A second class of tools that provide XML mapping capabilities offers a more functional, programming-language-oriented approach. Rather than directing the user to visually define relationships between a source and a target, the mapping tool assists the user in building the transformation logic, often through a spreadsheet metaphor. An example of this approach is the IBM WebSphere\* DataStage\* TX product. Source and target schemas are converted to a proprietary type system, and users fill in spreadsheet cells to create submappings that functionally map one type to another. Submappings can be nested into larger mappings to map more complex types.

The functional approach gives the user more control over specifying both the details and the order of the actual transformation steps and is typical of scenarios with a high degree of complexity and performance requirements. A function-based tool can offer quite powerful expressions and, depending on the language exposed to users, can offer Turing-machine-equivalent transformations.

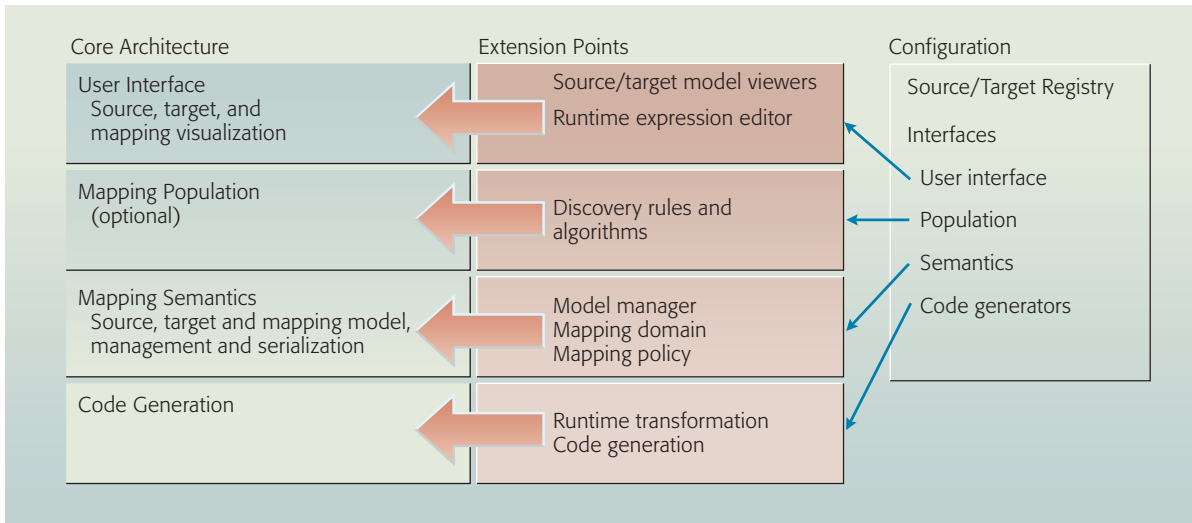
Function-based mapping tools are generally targeted at skilled technical users for two reasons. First, defining a map programmatically generally requires that the user learn and understand a programming language. Second, the power and fine-grained control of a function-based approach comes at the expense of visualization; it is easy to see the details of a specific source-to-target transformation, but it is more difficult to intuit a higher-level source-to-target mapping.

### **Discussion**

A more detailed review of XML mapping products reveals that there are a wide variety of tools available and that capabilities vary from vendor to vendor. Most are targeted at solving a particular use case (for example, converting relational data to XML, transforming one XML document to another, transforming one message format to another) and, more significantly, all are primarily focused on assisting users to generate executable transformation code. The popularity and number of such tools indicates that such code generation is very valuable. However, code generation is really only one aspect of building and maintaining business applications, particularly when such applications are focused on integrating existing applications to perform new function. We believe there are two additional goals that XML mapping technology can and should facilitate: mapping reuse in multiple contexts and treatment of the mapping information itself as metadata.

### **Mapping reuse**

A multistep, multicomponent application may require multiple mapping steps along the way. Consistency of structure, terminology, and function throughout the business application is critical and difficult to achieve with today's mapping tools. It is expensive to research, purchase, and learn how to use different mapping tools for different use cases, and there is often a disparity of function offered at each of the different steps of the application. For example, the standard banking model in our EII scenario should also serve as the canonical model for information in the SOA scenario, even though the executable transformation language and runtime systems differ. Ideally, the banking institution should use a single XML mapping tool to transform and structure its information throughout the enterprise, regardless of whether it is rendered in SQL/XML, XSLT, or even Java code.



**Figure 4**  
An extensible model-driven XML mapping architecture

### Mapping as metadata

Enterprises today must deal with thousands of information sources, and the majority of time on an integration exercise is spent trying to find the subset of information that is relevant. A significant integration challenge is to discover how and what to integrate in the first place and to remember why you did so later on, either because applications have evolved or because different team members at different geographic locations and different times are involved in building and maintaining the application. At its core, mapping information is metadata that captures the relationships among a set of information sources and documents the decisions made and reasons why the information was structured in a particular way.

For example, suppose a particular piece of information could potentially come from two sources (such as a credit rating for a customer that had accounts at two subsidiaries of the bank), and the data architect, in consultation with business analysts, determined which of multiple sources to use and how to combine the information. The mapping implicitly records not only the transformation, but the thought process behind the transformation, including which source was chosen (and which was not), or which computations were performed to combine the information.

Unfortunately, because XML mapping tools on the market today are focused on code generation, they

store the mapping information either in a proprietary format or simply as executable code itself. Neither format is suitable for mapping to be used as metadata. Very few tools offer any help at discovering relationships among sources. Ideally, mapping metadata should be exposed and structured in a way that facilitates a collaborative and iterative design process (including user-defined annotations to document decisions). Such mapping information can be used to record and document relationships, can be populated by automated discovery processes, and can be stored in a repository, where it can be reviewed, reused, and provide the ability to play “what if” scenarios and assess the impact of changes.

### AN EXTENSIBLE XML MAPPING ARCHITECTURE

We propose an extensible, model-based architecture for XML mapping technology that supports the goals just described. The architecture includes several of the ideas developed as part of the Clio research project<sup>12–14</sup> and also fits (at the high level) in the model management framework of Bernstein.<sup>15</sup> This architecture has been implemented and tested for a variety of source and target combinations—including relational database (RDB)/RDB, RDB/XML, XML/XML, Web Services Description Language (WSDL)/WSDL, and Java object/XML—and to generate a variety of transformation languages, including SQL, SQL Data Definition Language (DDL), XQuery, XSLT, and Java. It forms the foundation for IBM Rational\* Data Architect and has facilitated a



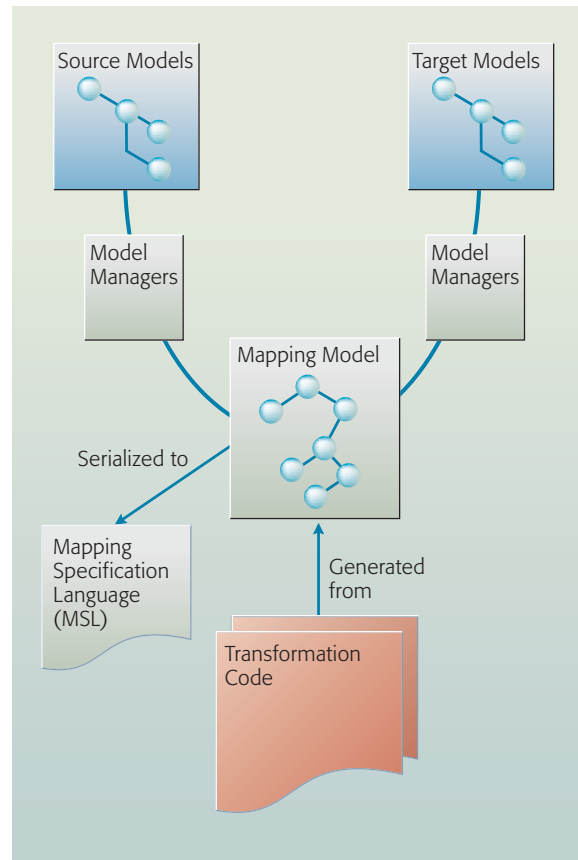
number of additional research projects in the areas of discovery,<sup>16</sup> schema evolution,<sup>17,18</sup> query re-write,<sup>19</sup> mapping semantics,<sup>20</sup> mapping composition,<sup>21</sup> and dependency extraction.<sup>22</sup>

As illustrated in *Figure 4*, the architecture is composed of four components that provide the core services for XML mapping: user interface, mapping population, mapping semantics, and code generation. The user interface component provides the functions to visually display, create, and edit a mapping. The mapping population component provides an interface to programmatically compute a mapping. The mapping semantics component captures and manages the semantics of the transformation. The code generation component translates the mapping semantics into one or more forms of executable transformation code. As shown in the middle column of the figure, each of these components includes appropriate extension points to tailor the base function to specific mapping scenarios. A *configuration* combines the base function and extensions with a registry to form a customized mapping tool. Each of the components is described in more detail later, followed by a description of the configurations that satisfy the requirements of the scenario introduced before. We begin with the mapping semantics component, which is the anchor point for all other components.

### Mapping semantics

As shown in *Figure 5*, a mapping exercise involves two kinds of models: models that represent the sources and targets of mappings and the mapping itself. Source and target models represent the structures (or *schemas*) to be transformed. They are typically pre-existing and well defined, and often have standards organizations governing their structure. As discussed in the introduction, for XML-based scenarios, at least one of the endpoints is defined by XML Schema. Integration scenarios often add additional formats. For example, the scenarios presented before require at least three other formats: relational schema, UML, and WSDL.

The mapping model captures a transformation between a set of sources and targets as declarative assertions at a higher level of abstraction than the target transformation language, and serves as the anchor for achieving the goals of code generation, mapping reuse, and mapping as metadata. First, it records the basic transformation semantics in a runtime-independent format, which can be written



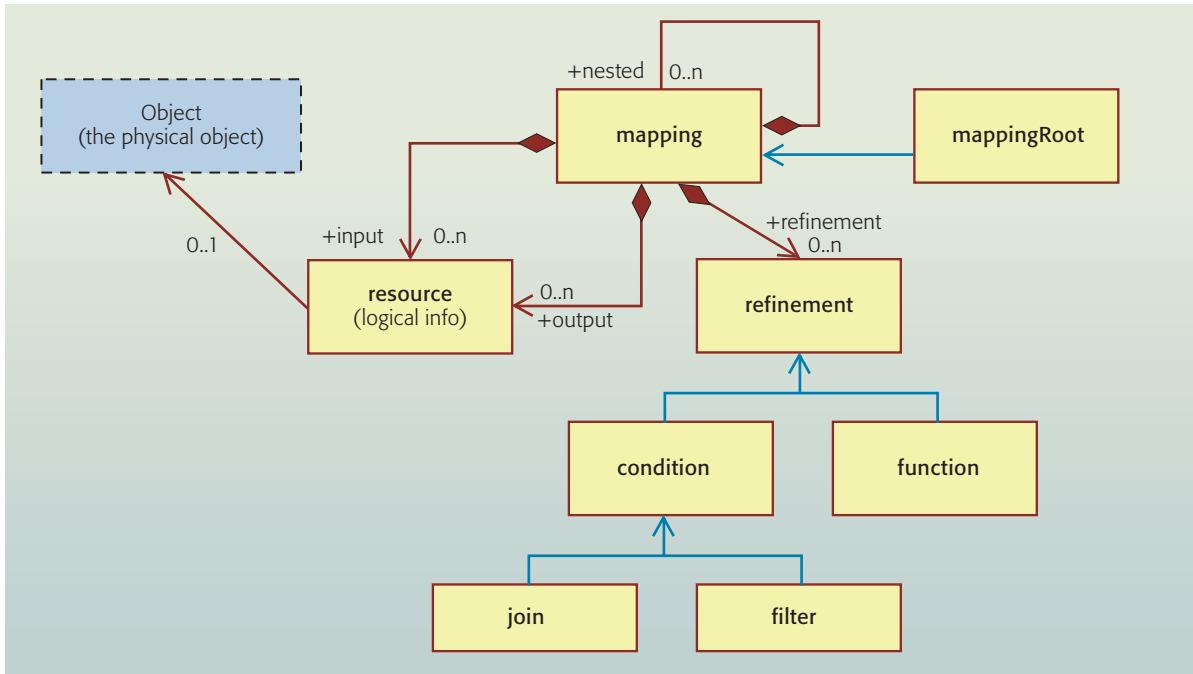
**Figure 5**  
Extensible model-driven mapping

or rewritten into multiple transformation languages. Second, the declarative assertions can be expressed formally as logical operators and so promote programmatic reasoning, such as composition, aggregation, inversion, and extraction of dependencies to convert a transformation from one runtime format to another. Finally, the mapping model can be annotated and serialized to a human-readable format that facilitates reporting and collaboration.

A detailed description of the mapping model first requires an introduction to *model managers*, whose role within the architecture is to handle schema model heterogeneity.

### Model managers

Figure 5 shows that model managers provide a virtual translation of the native model into a canonical, or common, model in which semantics can be defined and validated. For XML-centric mapping, XML Schema is a logical choice for the



**Figure 6**  
The mapping object model

canonical format. Each type of source or target model requires a model manager, which is made up of three extension points: a model viewer, a transformer, and a resolver.

The *model viewer* displays the model in its native (or most intuitive) format by the user interface.

The role of the *transformer* is to convert instances of the schema model into the canonical schema model. The model manager uses the transformer to maintain a bidirectional mapping between the original (and visualized) model instances and the canonical model instance. Validation and code generation rely on this translation to convert XML schema objects to the original format, and vice versa. The mapping model contains references (or endpoints) to the source and target models. These references are implemented with *resolvers*, whose role is to uniquely identify and define objects in the source and target models (such as an attribute name on an XML schema or a column name of a relational table).

### Mapping model

The mapping model consists of two components: an in-memory object model that can be visualized and

interpreted and a human-readable serialization XML format, called the Mapping Specification Language (MSL). The ability of the mapping model to be serialized is a key benefit of the architecture. It allows external processes to programmatically produce mapping information, to interpret the mapping instances into other formats (e.g., to create an HTML [Hypertext Markup Language] report of the mapping contents), and to persist it (give it permanent storage) in a shared metadata repository, where it can be further queried, searched, and analyzed.

*Figure 6* shows in UML the core classes that make up the mapping object model. Relationships between source and target objects are referred to as *mappings*, and an instance of the mapping model is a tree of *mapping objects*. The `mappingRoot` object represents the top-level mapping object. Each mapping may also have a list of refinements, or annotations, that add semantics to the relationship. The following are examples of refinements:

- **Scalar functions:** `concat (source.firstname, source.lastname) → target.fullname`
- **Conditions:** `if (source.date <= source.expiration_date)`

- **Set functions:** `union (source1, source2)`
- **User-defined annotations:** ‘This relationship was added by Jane Doe’

We can use a subset of the EII scenario to build an instance of a mapping model, which we will express in its serialized MSL format. We start by creating a `mappingRoot` object and setting its input and output resource list to include only one source-side resource (represented by the file URI `stdModel.uml`) and one target resource (represented by `bankTeller.xsd`), respectively.

```
<mappingRoot>
  <input path="stdModel.uml" root
    ="StandardModel"/>
  <output path="bankTeller.xsd" root
    ="BankTeller"/>
</mappingRoot>
```

We classify mapping into two groups: *value mappings* represent correspondences between schema elements that contain data values, and *containment mappings* represent mappings between the schema objects that contain the elements mapped by value mappings.

Value mappings are used to map one or more input-side elements into one output-side element. The following example captures two value mappings for the `BankTeller` example. The first is a value mapping between the `Customer_ID` field of `Customer` and the `ID` field of `Client`. The second value mapping represents a more complex value mapping. `Customer firstName` and `lastName` on the `Standard Model` must be combined by means of a scalar `concat()` function to form `Client Name` on the `Bank Teller Model`. Here, the scalar function is captured as a “refinement” of the value mapping, and it is assumed that `concat()` is a valid function in the target transformation language. Notice the use of variable names as parameters to the function. The mapping model allows binding a variable name to every path expression and using that variable name in place of the expression.

```
<mappingRoot>
  <input path="stdModel.xsd" root
    ="StandardModel"/>
  <output path="bankTeller.xsd" root
    ="BankTeller "/>
  <mapping>
```

```
    <input name="first"
      path="/StandardModel/Customer/Customer_ID"/>
    <output path="/BankTeller/Client/ID"/>
  </mapping>
  <mapping>
    <input name="first "
      path="/StandardModel/Customer/firstName"/>
    <input name="last"
      path="/StandartModel/Customer/lastName"/>
    <output path="/BankTeller/Client/Name"/>
    <function value="concat ($first, $last)"/>
  </mapping>
</mappingRoot>
```

The transformation semantics of a value mapping in isolation are unambiguous. However, multiple value mappings taken together can introduce ambiguity. In the example just presented, one interpretation would apply both value mappings simultaneously to produce one target `Client` record for every `Customer` record. Another interpretation would create a new `Client` record for every `Customer_ID` value and for every concatenation of first name and last name. This interpretation represents an outer union of the mapped values; half of the `Client` records would only have a value for the `ID` element, and the other half would only have a value for the `Name` element. Unambiguous relationships are especially important for code generation, so that the correct transformation code can be generated. We use containment mappings to express such unambiguous relationships.

Containment mappings provide the context under which set value mappings operate, and they are defined between the schema elements that contain the elements used in value mappings. (We note that containment mappings are an extension of the Clological mappings,<sup>13</sup> which have also been studied from a theoretical perspective<sup>20</sup> under the name source-to-target tuple dependencies.) Containment mappings represent logical constraints that all source data instances and all target data instances must satisfy and have the following form:

```
forall (source-side binding of variables)
  where (source-side predicate)
exists (target-side binding of variables)
  where (target-side predicate)
satisfies (value mappings and other containment
  mappings)
```

```

<mappingRoot>
  <input path="stdModel.xsd" root="StandardModel"/>
  <output path="bankTeller.xsd" root="BankTeller"/>
  <mapping>                                <!-- containment mapping -->
    <input name="x0" path="/StandardModel/Customer"/> <!-- forall -->
    <output name="y0" path="/BankTeller/Client"/> <!-- exists -->
    <mapping>                                <!-- nested value mapping -->
      <input path="$x0/Customer_ID"/>
      <output path="$y0/ID"/>
    </mapping>
    <mapping>                                <!-- nested value mapping -->
      <input name="first" path="$x0/firstName"/> <!-- satisfies clause -->
      <input name="last" path="$x0/lastName"/>
      <output path="$y0/Name"/>
      <function value="concat($first, $last)"/>
    </mapping>
  </mapping>
</mappingRoot>

```

**Figure 7**  
Simple customer-to-client mapping

The `forall` clause contains one or more variables bound to sequences of values from the source side using XPath expressions. An optional predicate that uses the source side variables can be added after the `forall` clause to filter the sequence member that participates in the mapping. Similarly, the `exists` clause contains a list of target-side bindings. The target-side predicate allows for the specification of target-side conditions that must hold in the mapping. Finally, the `satisfies` clause contains a set of contained value mappings.

Of the two possible interpretations for the value mappings listed earlier, the former is the more likely as it maintains the association of source data to the data produced in the target:

```

forall x0 ∈ /StandardModel/Customer
exists y0 ∈ /BankTeller/Client
satisfies (y0/ID = x0/Customer_ID and y0/Name =
           concat(x0/firstName, x0/lastName))

```

This constraint can be read as follows: for all `Customer` tuples (`x0`), there will be at least one tuple in `Client` (`y0`) for which its `ID` field contains the same value as the `Customer_ID` field of the `Customer` tuple, and for which its `Name` field is composed by concatenating the `firstName` and `lastName` fields of the `Customer` tuple. This interpretation is captured in the mapping model in *Figure 7*, where the variables `$x0` and `$y0` are used to refer to the repeating elements of the models.

The MSL fragment in *Figure 8* captures the complete mapping between the Standard Model and Bank Teller. Three repeating elements on the Standard Model (`Customer`, `Accounts`, and `Transactions`) must be joined to produce the nested `Client` and `Accts` structure on the Bank Teller Model. All customers should be mapped regardless of whether they have an account (i.e., outer-join semantics), but the Bank Teller Model should include only accounts of type `Savings` or `Checking`.

Lines 5–16 map the top-level target structure `Client`, using only data from `Customer`. All `Customer` information will be mapped to `Client`, regardless of the other conditions in the mapping, accomplishing the outer-join semantics we wanted. Lines 17–43 show a containment mapping nested within the top-level containment mapping. This mapping takes advantage of the records already bound by the top level to express a join. In particular, line 20 joins the source `Customer` record bound to `$s0` with all the records in `Account` for which `Customer_ID` is the same. Line 21 adds another filtering condition to select only savings or checking accounts from the records selected from `Accounts`. Lines 22–29 are the value mappings associated with this nested mapping. Finally, lines 30–42 add another nested mapping within the second to map the `Transactions` information.

This mapping model captures the complete transformation of the Standard Model to the Bank Teller

```

1. <mappingRoot>
2.   <input path="stdModel.xsd" root="StandardModel"/>
3.   <output path="bankTeller.xsd" root="BankTeller"/>
4.   <mapping>
5.     <input name="s0" path="/StandardModel/Customer"/>
6.     <output name="t0" path="/BankTeller/Client"/>
7.     <mapping>
8.       <input path="$s0/Customer_ID"/>
9.       <output path="$t0/ID"/>
10.    </mapping>
11.    <mapping>
12.      <input name="first" path="$s0/firstName"/>
13.      <input name="last" path="$s0/lastName"/>
14.      <output path="$t0/Name"/>
15.      <function value="concat($first, $last)"/>
16.    </mapping>
17.    <mapping>
18.      <input name="s1" path="/StandardModel/Accounts"/>
19.      <output name="t1" path="$t0/Accts"/>
20.      <condition value="$s0/Customer_ID = $s1/Customer_ID"/>
21.      <condition value="$s1/Account_Type='Savings' or $s1/Account_type='Checking'"/>
22.      <mapping>
23.        <input path="$s1/AccountID"/>
24.        <output path="$t1/AcctID"/>
25.      </mapping>
26.      <mapping>
27.        <input path="$s1/Account_Type"/>
28.        <output path="$t1/AcctType"/>
29.      </mapping>
30.    </mapping>
31.    <mapping>
32.      <input name="s2" path="/StandardModel/Transactions"/>
33.      <output name="t2" path="$t1/Activity"/>
34.      <condition value="$s2/Account_ID = $s1/Account_ID"/>
35.      <mapping>
36.        <input name="$s2/Date"/>
37.        <output name="$t2/Date"/>
38.      </mapping>
39.      <mapping>
40.        <input name="$s2/Amount"/>
41.        <output name="$t2/Amount"/>
42.      </mapping>
43.    </mapping>
44.  </mapping>
45. </mappingRoot>

```

**Figure 8**  
Complete standard model to bank teller mapping

model, and converting it into a transformation language, such as XQuery, is straightforward. Two other fundamental concepts that must be captured by the mapping model are grouping and collating. This is particularly important in hierarchical models like XML, where grouping of data that shares a common parent value (or values) is naturally captured by the hierarchy of data. A full description

of how we model these operations in our model is beyond the scope of this paper.

### **Validation**

As previously noted, a key design point of the mapping model is to tolerate heterogeneity with respect to source and target models and transformation languages. The mapping semantics component contains a *validator* to determine if the

mapping model represents a valid transformation with respect to the source and target models and runtime transformation language. The validator serves two roles. First, it provides the context in which a mapping can be defined. *Context validation* is implemented as a context-sensitive consultation between the user interface and the validator to help guide the user to construct correct mapping models by providing input to the user interface with regard to which actions are valid in a particular context. Second, it provides *instance validation*, the logic to validate a mapping as it is defined or to validate a mapping after it has been reloaded.

The validator is customized for a given mapping scenario by two extension points: mapping domains and mapping policies. A *mapping domain* defines valid combinations of source and target models and transformation languages, and valid types and expressions based on these combinations. It consists of a *registry* that provides context validation to restrict the type of models that can be mapped and the type of transformation languages to generate, and it provides implementations for type and expression checkers that provide instance validation for particular mapping combinations. RDB/RDB→SQL, RDB/XML→SQL/XML, XML/XML→XSLT, and XML/XML→XQuery are examples of registry entries. A mapping domain that includes the RDB/XML→SQL/XML registry entry, for example, would also provide a type and expression checker based on SQL/XML. The type checker might verify that a date field has the appropriate cast function to map to an integer, whereas the expression checker might verify that a function `concat($first, $last)` is a valid SQL expression (after the variables are replaced by column names).

A *mapping policy* provides context-sensitive input to the user interface to guide a user when interactively defining new mappings between objects. A mapping policy specifies types of endpoints that are valid for value or containment mappings (e.g., source elements that can appear in the `forall` clause and target elements that can appear in the `exists` clause), as well as refinements that are valid for a given mapping object. For example, for XML schemas, elements that represent repeating structures are valid endpoints for containment mappings, whereas elements that can contain data elements are valid endpoints for value mappings. For an RDB/XML→SQL/XML mapping domain, column objects

are valid endpoints for value mappings, and table objects are valid endpoints for containment mappings. SQL scalar functions are valid for value mappings, and filters, joins, and aggregate functions (including the SQL/XML functions) are valid for containment mappings.

### User interface

The user interface is a key component that allows users to visualize, create, and modify the relationships and semantics that make up a mapping. Earlier we observed that for existing tools, mapping visualization metaphors are either relationship based or function based, and that relationship-based tools typically offer a higher level of abstraction at the expense of expressive power, while function-based tools offer more expressive power but require a higher degree of skill on behalf of the user.

Thus, both metaphors offer strong support for code generation, each providing a different set of advantages to the user. However, because of the higher degree of abstraction and relationship visualization, we believe a relationship-based metaphor is better suited for the goals of managing complexity and mapping as metadata. We have derived a core set of user interface components from that metaphor, with appropriate extension points to support heterogeneous source and target models, multiple transformation languages, and views that render the mapping in domain-specific metaphors (including, for example, a function-based approach).

The user interface consists of source and target views, a mapping view, an outline view, and a properties view. The source and target views render source and target models as tree views, and the mapping view visually displays relationships between schema elements by interpreting the mapping model. The mapping view allows users to draw or annotate relationships with additional semantics, such as functions, conditions, or user-defined annotations, and provides a visual cue to denote the presence of such annotations. The outline view provides a more compact, text-based view of the mappings.

Properties of source, target, and mapping objects can be viewed and edited in the properties view. Our architecture includes large schema support, including features to filter elements or subtrees from the source and target views and to hide elements that do



not participate in the mapping. With this support, users can work with sparse mappings involving large schemas, which are quite common. In addition, to focus a user's attention when visually manipulating large and complex mappings, the user interface includes the concept of *visual partitions* in the mapping view, which filter the visualization into logically related mapping groups. Such groups can be automatically computed, such as containment mappings and their nested value mappings, or defined by users, as is provided today by Microsoft BizTalk\*\*.

Extensibility at the user interface is captured in three aspects: source and target model viewers, transformation language-specific editors, and additional domain-specific views. A model viewer is a prebuilt or custom-built renderer to display a particular kind of model in either the source or target panel as a tree of objects. The user interface allows such viewers to be mixed, based on the particular mapping scenario (for example, RDB→XML or RDB→UML). Although extensions to the mapping model specific to the transformation language can be specified textually in the properties view, an optional transformation language editor can be plugged in to build more complex expressions. For example, for an RDB→XML mapping configuration that generates SQL/XML statements, the user interface can be extended with an editor that assists a user to build and validate SQL/XML expressions. Similarly, an XML→XML mapping interface may include an XSLT or XQuery expression builder. Finally, a relationship-based metaphor may not be the metaphor of choice for all mapping scenarios. The model-centric approach makes it possible to extend (or replace) the user interface with a domain-specific view, such as the function-based approach described in the section "State of the art."

### Mapping population

The user interface enables users to create and modify mapping information. However, as shown in the EII scenario, manually specifying mapping information is, at best, a tedious process and, at worst, not possible if the relationships are not easily identifiable. The mapping population component is an optional interface to programmatically populate the mapping model with relationships that may be inferred from a variety of techniques, including structural comparison (the source `name` element has

two attributes and the target `cust_name` element has two attributes), data comparison (samples of data for source element `ssn` and target element `taxid` have overlapping values), and semantic analysis (`customer` and `client` refer to similar concepts).

Programmatic-mapping discovery and population is important for the goal of mapping as metadata, particularly when integrating large and complex schemas. Even if not all discovered relationships are correct, they very often draw attention to where

■ Mapping tools follow one of two visual metaphors, either highlighting relationships between a source and target or highlighting functional steps required to transform a source to a target ■

models overlap. Extensibility in this component is crucial, as very often it is possible to construct highly accurate, domain-specific matching techniques, such as algorithms that match chemistry structures for a life-science application. Custom discovery rules and algorithms can record inferred relationships in the mapping model (along with confidence values and supporting evidence), which can then be visualized, persisted, and used for later analysis.

The mapping population component exposes a simple interface that takes as input a set of source and target model elements and an existing mapping-model instance, then returns a populated mapping model. The mapping population interface enables algorithms to be dynamically combined such that independent algorithms reinforce the results of each other. For example, semantic-name matching might suggest that `customer_taxid` and `client_ssn` are similar concepts, and a data signature analysis may confirm that the data in both columns follows the same pattern. Alternatively, an edit-distance analysis may suggest a strong correlation between `eaddress` and `address`, but a regular expression analysis of the data shows two very different patterns (e-mail addresses as opposed to postal addresses).

Our implementation supports a set of algorithms applicable to many domains. These include a dependency extraction technique,<sup>22</sup> five metric algorithms, and two matching algorithms. The metrics include edit-distance name similarity, thesaurus-based semantic name matching,<sup>23</sup> data signature comparison,<sup>16</sup> and data distribution analysis.<sup>24</sup> The matching algorithms include a simplistic greedy approach and a more measured “stable marriage” approach.<sup>25</sup>

### Code generation

Code generation is the process of rendering a mapping-model instance to a target transformation language. It is accomplished with two components: a navigator and a generator. The *navigator* visits the target schema model, using the canonical representation created and maintained by the model manager. For each target structure to be generated, the navigator produces the set of mapping-model objects required to build the structure and passes it to the generator. The *generator* then outputs the corresponding transformation logic in the appropriate language. The mapping architecture provides a common navigator and allows for diverse generators to attach themselves to produce the transformation code. Our implementation of this architecture includes generators for XQuery, XSLT, SQL, and SQL/XML.

For example, consider an XQuery generator. Figure 1 shows an XQuery that is generated from the `StandardModel` to `BankTeller` mapping of Figure 2. For each target schema object associated with a containment mapping, the code generator creates a `for loop` over the source to retrieve the records that will be used to construct the target element. Filters and join conditions associated with the containment mapping are placed in the corresponding `where` clause of the `for loop`. Target schema objects nested within the containment mapping are either generated by value mappings or other containment mappings. Constructors for the elements are placed in the `for loop` for those generated by value mappings, and nested `for loops` are added to the `for loop` for those generated by other containment mappings.

### Customizing XML mapping architecture

In this section, we show how to customize a mapping tool, based on this architecture, to support the EII and SOA scenarios. Note the overlap between

the EII and SOA configurations—by sharing a common architecture, both scenarios benefit from common source and target viewers, mapping population, and a shared mapping model that can be transformed into SQL/XML or XSLT statements.

As shown in *Table 1*, a custom instance of a mapping tool can be dynamically customized for each scenario through a configuration (specified in a file). A *configuration* is made up of a source/target registry that defines the valid source/target→transformation language combinations and a set of packages that provide the implementation for the extension points.

Table 1 shows the source/target registries for the EII and SOA scenarios. The EII scenario shows four valid source and target combinations: RDB/RDB, RDB/UML, RDB/XML, and UML/XML, with target languages of SQL and SQL/XML. The SOA scenario adds XML/XML and WSDL/WSDL combinations and, in both cases, the target language is assumed to be XSLT.

The user interface extensions include existing model viewers for RDB, UML, XML, and WSDL. The XML viewer can be shared for both the EII and SOA scenarios. The EII configuration includes a visual SQL expression builder to assist the user to build SQL-compatible mapping refinements, and the SOA configuration includes a visual XQuery expression builder for XSLT-compatible refinements. Both scenarios benefit from an implementation of discovery-assisted mapping-model population based on semantic name matching.<sup>23</sup>

The EII scenario requires model managers for RDB, UML, and XML Schema. The SOA scenario can share the XML Schema model manager and also requires a WSDL model manager.

Recall that a mapping domain is composed of a source/target registry (defined in the configuration file) and type and expression checkers. We were able to take advantage of existing parsers for SQL and SQL/XML for the EII mapping domain, and existing XSLT parsers for the SOA mapping domain for type and expression checking by translating mapping objects into small code fragments and then invoking the appropriate parser for validation. For

**Table 1** Mapping configuration for EII and SOA

	EII Configuration	SOA Configuration
<b>Source/Target Registry</b>	RDB/RDB→SQL RDB/UML→SQL RDB/XML→SQL/XML UML/XML→SQL/XML	XML/XML→XSLT WSDL/WSDL→XSLT
<b>User Interface</b>	com.ibm.core.rdb.viewer com.ibm.core.uml.viewer com.ibm.core.xml.viewer <sup>†</sup> com.ibm.core.rdb.exp	com.ibm.core.xml.viewer <sup>†</sup> com.ibm.core.wSDL.viewer com.ibm.core.xslt.exp
<b>Population</b>	com.ibm.ext.discovery.semname <sup>†</sup>	com.ibm.ext.discovery.semname <sup>†</sup>
<b>Semantics Model Manager</b>	com.ibm.ext.rdb.modelmgr com.ibm.ext.uml.modelmgr com.ibm.ext.xml.modelmgr <sup>†</sup>	com.ibm.ext.xml.modelmgr <sup>†</sup> com.ibm.ext.wSDL.modelmgr
<b>Mapping Domain</b>	com.ibm.ext.rdbumlxml.mapdomain	com.ibm.ext.xml.mapdomain com.ibm.ext.wSDL.mapdomain
<b>Mapping Policy</b>	com.ibm.ext.rdb.mappolicy com.ibm.ext.uml.mappolicy com.ibm.ext.xml.mappolicy <sup>†</sup>	com.ibm.ext.xml.mappolicy <sup>†</sup> com.ibm.ext.wSDL.mappolicy
<b>Code Generation</b>	com.ibm.ext.sqlxml.gen	com.ibm.ext.xquery.gen

<sup>†</sup>Indicates packages that are shared between the two components

example, consider the following refinement on a value mapping for the EII scenario:

```
concat ($customer/firstname, $customer/
lastname)
```

The RDB model manager resolves `$customer/firstName` and `$customer/lastName` expressions to be the `firstName` and `lastName` columns of the Customer. The SQL (and SQL/XML) validator converts the refinement on the mapping to the following query:

```
SELECT concat (C.firstName, C.lastName) FROM
StandardModel.Customer C,
```

This query fragment is then passed to the SQL parser for validation.

The RDB mapping policy is straightforward, specifying that columns are valid endpoints for value mappings and table objects are valid endpoints for containment mappings. The XML Schema mapping policy is more complex. Valid containment-mapping endpoints are repeatable XML elements, and any

XML Schema object that represents a data value is a valid value-mapping endpoint. This includes XML attributes and XML elements with primitive data types. However, this also includes XML elements with complex types that allow *mixed content*—meaning that instances of this XML element can have text child nodes. For example, consider this XML fragment:

```
<part><id>x003</id>notebook</part>.
```

Here `part` has a complex type because it can contain `id` elements within. But `part` also allows mixed content, in this case, the `notebook` text within. These text nodes are mappable data values and should be allowed as valid value-mapping endpoints.

The SOA scenario requires a code generator for XSLT. The EII scenario requires code generators for SQL and SQL/XML. SQL/XML is a superset of SQL with additional functions to tag relational data as either XML elements or XML attributes and to aggregate multiple XML elements into a sequence of elements. As a result, it is possible to use one code generator to generate both SQL and SQL/XML.

## RELATED WORK AND REMAINING CHALLENGES

A number of new challenges for XML mapping technology arise from the natural progression of XML to a mature technology. The following are important related or open issues: schema matching; round tripping and dependency extraction (e.g., between mappings and generated artifacts); schema evolution, including migration, adaptation, and mapping inversion; sequential and parallel mapping composition; and schema integration.

*Schema matching*—Schema matching is the automatic discovery of correspondences between heterogeneous data sources. Examples of schema matching research include References 16 and 26–33.

*Round tripping and dependency extraction*—The common architecture proposed earlier provides a solution to quickly generate a mapping between two formats and render it as executable code. The generated executable code can be further modified by humans (for example, XSLT experts) or other applications, and such modifications can happen concurrently with edits on the mapping model that created the artifact. *Round tripping*, or synchronizing these modifications to keep the mapping model synchronous with the runtime code, is an important open question.

We referred earlier to *dependency extraction*, or extracting a mapping specification from existing legacy transformation code (e.g., XSLT, SQL, COBOL). Initial results<sup>22</sup> are very promising, but a number of important challenges remain. In particular, our mapping specification is not a general-purpose transformation language, and thus, expressiveness problems arise when attempting to decode expressions in Turing-complete languages.

*Schema evolution—migration, adaptation, and inversion*—Changes to the source and target structures, or *schema evolution*, is often inevitable, particularly with XML data, where it is easy to change the way data is structured or tagged. Schema evolution is almost always associated with *schema migration*, which involves mapping data that conforms to one version of a schema to a new (evolved) schema. The XML mapping technology discussed in this paper facilitates schema migration.

A second challenge in schema evolution is to devise methods to automatically or semiautomatically maintain and adapt mappings and their corre-

sponding executable transformation code in the face of schema evolution. *Mapping adaptation* is the process of migrating those existing mappings and the corresponding generated executable code into new mappings and code that refer to new or changed schema elements and preserve as much of the logic in the old mapping as possible. Velegrakis, Miller, and Popa<sup>17</sup> study mapping adaptation when an edit script that captures the changes in the source or target schema is available. In this approach, incremental changes are made to the mapping for each change in the schemas. In the more recent work of Yu and Popa,<sup>18</sup> an edit script is not needed to adapt mappings. Instead, a new mapping is created (discovered or given by the user) between the old version and the new version of the evolved schemas. This evolved mapping is composed with the previous mapping to derive the adapted mapping.

*Mapping inversion* refers to generating a mapping whose transformation semantics is the inverse of a given mapping. Ideally, applying the inverse mapping should yield back the original data. Mapping inversion is a fundamental challenge that typically implies more than reversing the direction of the mapping. For example, a mapping that involves the union of two data sets cannot be inverted without loss of information unless additional bookkeeping is kept at runtime to capture data lineage. For those cases in which information loss may be acceptable, defining a correctness criterion for the inverse that reflects the minimal acceptable information loss is an open problem and would be quite useful.

*Mapping composition (sequential and parallel)*—As illustrated by our earlier scenarios, sequential and parallel mapping composition are important challenges to address. *Sequential composition* refers to the problem of transitively combining two successive mappings (from a schema S1 to a schema S2, and from the schema S2 to a schema S3) into a single mapping (from S1 to S3) that, in terms of transforming the data, has the same effect. Such composition can eliminate intermediate steps in a transformation process, for example, such as occurs when the custom views are created from the standard model in the EII example of Figure 2. Sequential composition also enables mapping reuse. For example, mappings are frequently established between two logical models and later applied to two

physical schemas that are not identical to the logical models, but do themselves have mappings to or from the logical models.

*Parallel composition* refers to deriving an aggregate map from multiple sources into a common target, based on a set of existing individual mappings from each source to the target. A common goal of such an aggregate map is to merge data from multiple sources with overlapping information. Note that this requires more than just a simple union of the mappings; parallel composition must typically take into account additional information relating the sources, such as intersource constraints, or more generally, intersource mappings and additional properties of the target schema, such as key or functional dependencies.

Preliminary solutions on sequential mapping composition and mapping adaptation already exist for a subset of mapping expressions.<sup>17,18,21,34</sup> It remains to be seen how those solutions can be extended to more general cases and whether they can be made fully scalable and usable in industrial-strength tools. The work of Nash et al.<sup>35</sup> focuses on composition in a mapping language that is more general (it allows for bidirectional mappings), but, at the same time, simpler than the proposed mapping model as it only covers the relational case. Their results so far are mostly theoretical. Parallel composition is an open problem.

*Schema integration*—Schema integration often appears when there is no *a priori* target schema, and instead, such a representation must be derived from the sources. Ideally, the target schema should be able to represent all the information from the source schemas with minimal duplication (i.e., elements that come from different source schemas but represent the same data should be merged). Such duplicate elimination is nontrivial, as it must take into account the complex relationships (mappings or constraints) that may exist at the data level between the source schemas. Most schema integration work has focused on using syntactic correspondences between the source schemas (e.g., see Reference 36). A proposal for using mappings among source schemas to construct a target schema appears in the metadata model management framework of Bernstein.<sup>15</sup> An XML schema integration system is described by Sakamuri et al.<sup>37</sup>

## SUMMARY

In this paper, we discussed how the importance of mapping and transformation has increased as enterprises select XML as their underlying technology for integration. We described two scenarios that demonstrate how XML mapping is used to facilitate integration, and we drew from these scenarios the requirements for mapping that go beyond what is available today. A key insight derived from the

■ The mapping model consists of two components: an in-memory object model that can be visualized and interpreted and a human-readable serialization XML format, called the Mapping Specification Language ■

discussion is that enterprise integration applications are typically multistep processes built from a variety of existing components with overlapping function, and a majority of development time is spent trying to find those components and identify and exploit the relevant overlap as a means for integration. We observed that XML mapping can help to automate this process by capturing, recording, and reusing metadata about the integration activity itself, and we described an extensible model-based architecture that evolves the current state-of-the-art mapping tools from functioning primarily as point-in-time code-assistance tools to support the equally important goals of reusability and mapping as metadata. Finally, we noted that XML mapping is a technology that falls into a broader research context that intersects both the XML and integration topics. We identified a rich set of challenges that remain in schema evolution, composition, and schema integration, and we have demonstrated that the model-based architecture proposed here can be used as a platform to further explore these areas.

\*Trademark, service mark, or registered trademark of International Business Machines Corporation.

\*\*Trademark, service mark, or registered trademark of Object Management Group, Inc., Sun Microsystems Inc., Progress Software Corporation, or Microsoft Corporation in the United States, other countries, or both.



## DISCLAIMER

The examples in this paper are provided on an as-is basis. They may be copied and modified in any form without payment to IBM for the purposes of designing and developing application programs. These examples have been tested, but they have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these examples in your particular environment. Use these examples as models for your own situations.

## CITED REFERENCES

1. Enterprise Information Integration, <http://www.dmreview.com/portals/portal.cfm?topicId=230290>.
2. New to SOA and Web Services, IBM Corporation, <http://www-128.ibm.com/developerworks/webservices/newto/>.
3. SQL/XML, SQLX.org, <http://www.sqlx.org/SQL-XML-resources/SQL-XML-resources.html>.
4. *XSL Transformations (XSLT), Version 1.0*, James Clark, Editor, W3C Recommendation (November 16, 1999), <http://www.w3.org/TR/xslt>.
5. *XQuery 1.0: An XML Query Language*, S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, Editors, W3C Candidate Recommendation (November 3, 2005), <http://www.w3.org/TR/xquery/>.
6. *Xalan-Java Version 2.7.0*, The Apache XML Project, <http://xml.apache.org/xalan-j/>.
7. Galax, <http://www.galaxquery.org/>.
8. WebSphere Software, IBM Corporation, <http://www-306.ibm.com/software/websphere/sw-bycategory/subcategory/SWB50.html>.
9. Metamatrix Server™, [http://www.metamatrix.com/pages/products/mm\\_enterprise.htm](http://www.metamatrix.com/pages/products/mm_enterprise.htm).
10. iWay, Sify Ltd., <http://www.iway.com/home.html>.
11. *Unified Modeling Language (UML), Version 2.0*, <http://www.omg.org/technology/documents/formal/uml.htm>.
12. R. J. Miller, L. Haas, and M. Hernandez, "Schema Mapping as Query Discovery," *Proceedings of the 26th International Conference on Very Large Data Bases*, Cairo, Egypt (2000), pp. 77–88.
13. L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin, "Translating Web Data," *Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China (2002), pp. 598–609.
14. L. M. Haas, M. A. Hernandez, H. Ho, L. Popa, and M. Roth, "Clio Grows Up: From Research Prototype to Industrial Tool," *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data*, Baltimore, MD (2005), pp. 805–810.
15. P. A. Bernstein, "Applying Model Management to Classical Meta Data Problems," *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA (2003), pp. 209–220.
16. F. Naumann, C.-T. Ho, X. Tian, L. M. Haas, and N. Megiddo, "Attribute Classification Using Feature Analysis," *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA (2002), p. 271.
17. Y. Velegrakis, R. J. Miller, and L. Popa, "Preserving Mapping Consistency under Schema Changes," *The VLDB Journal* **13**, No. 3, 274–293 (September 2004).
18. C. Yu and L. Popa, "Semantic Adaptation of Schema Mappings When Schemas Evolve," *Proceedings of the 31st International Conference on Very Large Data Bases*, Trondheim, Norway (2005), pp. 1006–1017.
19. C. Yu and L. Popa, "Constraint-Based XML Query Rewriting for Data Integration," *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, Paris, France (2004), 371–382.
20. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, "Data Exchange: Semantics and Query Answering," *Theoretical Computer Science* **336**, No. 1, 89–124 (2005).
21. R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan, "Composing Schema Mappings: Second-Order Dependencies to the Rescue," *Proceedings of the ACM Symposium on Principles of Database Systems*, Paris, France (2004), pp. 83–94.
22. A. Fokoue, "Extracting Input/Output Dependencies from XSLT 2.0 and XQuery 1.0," *Proceedings of the Extreme Markup Languages Conference*, Montreal, Quebec, Canada (2005), <http://www.mulberrytech.com/Extreme/Proceedings/html/2005/Fokoue01/EML2005Fokoue01.html>.
23. D. Caragea and T. F. Syeda-Mahmood, "Semantic API Matching for Automatic Service Composition," *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, New York (2004), pp. 436–437.
24. P. Brown and P. J. Haas, "BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data," *Proceedings of the 29th International Conference on Very Large Databases*, Berlin, Germany (2003), pp. 668–679.
25. D. Gale and L. S. Shapely, "College Admissions and the Stability of Marriages," *American Mathematical Monthly* **69**, 9–14 (1962.)
26. W.-S. Li and C. Clifton, "Semint: A System Prototype for Semantic Integration in Heterogeneous Databases," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA (May 1995), p. 484.
27. A. Doan, P. Domingos, and A. Y. Halevy, "Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA (2001), pp. 509–520.
28. J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic Schema Matching with Cupid," *Proceedings of the 27th International Conference on Very Large Data Bases*, Rome, Italy (2001), pp. 49–58.
29. S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching," *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA (2002), pp. 117–128.
30. J. Kang and J. F. Naughton, "On Schema Matching with Opaque Column Names and Data Values," *Proceedings of the 22nd ACM SIGMOD/PODS International Conference on Management of Data*, San Diego, CA (2003), pp. 205–216.
31. G. Shah and T. F. Syeda-Mahmood, "Searching Databases for Semantically Related Schemas," *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Sheffield, United Kingdom (2004), pp. 504–505.



32. P. Brown, P. J. Haas, J. Myllymaki, H. Pirahesh, B. Reinwald, and Y. Sismanis, "Toward Automated Large-Scale Information Integration and Discovery," in *Data Management in a Connected World*, T. Härder and W. Lehner, Editors, Springer-Verlag Publishing, Heidelberg, Germany (2005).
33. A. Bilke and F. Naumann, "Schema Matching Using Duplicates," *Proceedings of the 21st International Conference on Data Engineering*, Tokyo, Japan (2005), pp. 69–80.
34. J. Madhavan and A. Y. Halevy, "Composing Mappings among Data Sources," *Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany (2003), pp. 572–583.
35. A. Nash, P. A. Bernstein, and S. Melnik, "Composition of Mappings Given by Embedded Dependencies," *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Baltimore, MD (2005), pp. 172–183.
36. R. A. Pottinger and P. A. Bernstein, "Merging Models Based on Given Correspondences," *Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany (2003), pp. 826–873.
37. B. C. Sakamuri, S. K. Madria, K. Passi, E. Chaudhry, M. K. Mohania, and S. S. Bhowmick, "AXIS: An XML Schema Integration System," *Proceedings of the 22nd International Conference on Conceptual Modelling (ER'03)*, Chicago, IL (2003), pp. 576–578.

Accepted for publication October 12, 2005.

Published online May 10, 2006.

#### **Mary Roth**

IBM Almaden Research Center, 555 Bailey Road, San Jose, California 95141 ([torkroth@us.ibm.com](mailto:torkroth@us.ibm.com)). Ms. Roth is a software architect on the WebSphere Information Integration Solutions (WIIS) team. She has led several efforts in WIIS, including heterogeneous federation, discovery-driven integration design, and mapping technology for information integration. As a staff researcher, she contributed key advances in heterogeneous data integration techniques and federated query optimization, led efforts to implement federated database support in DB2, and contributed significantly to the ANSI/ISO SQL-MED standard. Ms. Roth has a B.S. degree in mathematics from Marquette University and an M.S. degree in computer sciences from the University of Wisconsin.

#### **Mauricio A. Hernandez**

IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 ([mauricio@almaden.ibm.com](mailto:mauricio@almaden.ibm.com)). Dr. Hernandez is a senior software engineer. His research interests include schema mapping, data integration, and data cleansing. Dr. Hernandez has a Ph.D. in computer science from Columbia University.

#### **Phil Coulthard**

IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 ([coulthar@ca.ibm.com](mailto:coulthar@ca.ibm.com)). Mr. Coulthard is a software architect and development manager. He is currently working on WebSphere Integration Developer, an integrated development environment for building SOA applications in which service integration is the key concern. He previously worked as developer, team lead, manager, lead architect, and customer advocate in the iSeries™ tools area of the Toronto Laboratory.

Mr. Coulthard is the coauthor of *Java for RPG Programmers* and *Java for S/390 and AS/400 COBOL Programmers*, from IBM Press.

#### **Lingling Yan**

IBM Almaden Research Center, 555 Bailey Road, San Jose, California 95141 ([llyan@us.ibm.com](mailto:llyan@us.ibm.com)). Dr. Yan is the lead developer for the mapping technology in Rational Data Architect. She holds a Ph.D. degree in computer science from the University of Alberta, Canada. Her interests include discovery algorithms and infrastructure, usability solutions and complexity management, and all means and channels of turning the value of mapping technologies into customer value.

#### **Lucian Popa**

IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 ([lucian@almaden.ibm.com](mailto:lucian@almaden.ibm.com)). Dr. Popa is a research staff member. He is one of the technical leaders of the IBM Clio project on schema mappings. He received an M.S. degree in computer science from the Polytechnic University of Bucharest and a Ph.D. degree in computer science from the University of Pennsylvania. Dr. Popa's research contributions have opened up a new area, called *data exchange*, of theoretical and applied research in information integration, now actively pursued by the research community.

#### **Howard (Ching-Tien) Ho**

IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 ([ho@almaden.ibm.com](mailto:ho@almaden.ibm.com)). Dr. Ho received a B.S. degree in electrical engineering from the National Taiwan University, and M.S., M.Phil., and Ph.D. degrees in computer science from Yale University. His past research interests include online analytical processing, communications issues for interconnection networks, algorithms for collective communications, graph embeddings, fault tolerance, and parallel algorithms and architectures. Dr. Ho's current research interests include XML and database languages, especially in schema mapping and data mining.

#### **Craig C. Salter**

IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 ([csalter@ca.ibm.com](mailto:csalter@ca.ibm.com)). Mr. Salter is the team leader of the Rational XML Web Services team. He received a B.Sc. degree in computer science from McMaster University, Canada. He and his team develop XML tools for the Rational Studio family of products with an emphasis on XML standard technologies supporting Web services. Mr. Salter is also a component lead on the (open source) Eclipse™ Web Tools Platform project. ■