Integrated data mapping for a software meta-tool

Jun Huh[1], John Grundy[1,2], John Hosking[1], Karen Liu[1], Robert Amor[1]
[1]Department of Computer Science and [2]Department of Electrical and Computer Engineering
University of Auckland
Private Bag 92019, Auckland 1142
New Zealand
jhuh003@ec.auckland.ac.nz,{john-g,john,karen,trebor}@cs.auckland.ac.nz

Abstract

Complex data mapping tasks often arise in software engineering, particularly in code generation and model transformation. We describe Marama Torua, a tool supporting high-level specification and implementation of complex data mappings. Marama Torua is embedded in, and provides model transformation support for, our Eclipse-based Marama domain-specific language meta-tool. Developers can quickly develop stand alone data mappers and model translation and code import-export components for their tools. Complex data schema and mapping relationships are represented in multiple, high-level notational forms and users are provided semi-automated mapping assistance for large models. MaramaTorua is a set of Eclipse plug-ins allowing close integration with other tools such as schema browsers, and with the Marama meta-tool itself.

1. Introduction

Many situations require the translation of data from one format to another. This includes:

- integrating two systems with different data file formats e.g. Computer-Aided Design tools;
- exchanging business messages between systems with differing protocols;
- transforming an XML document e.g. a UML diagram to SVG for rendering;
- generating code from a high-level model in model-driven development e.g. XMI model to Java or C#.

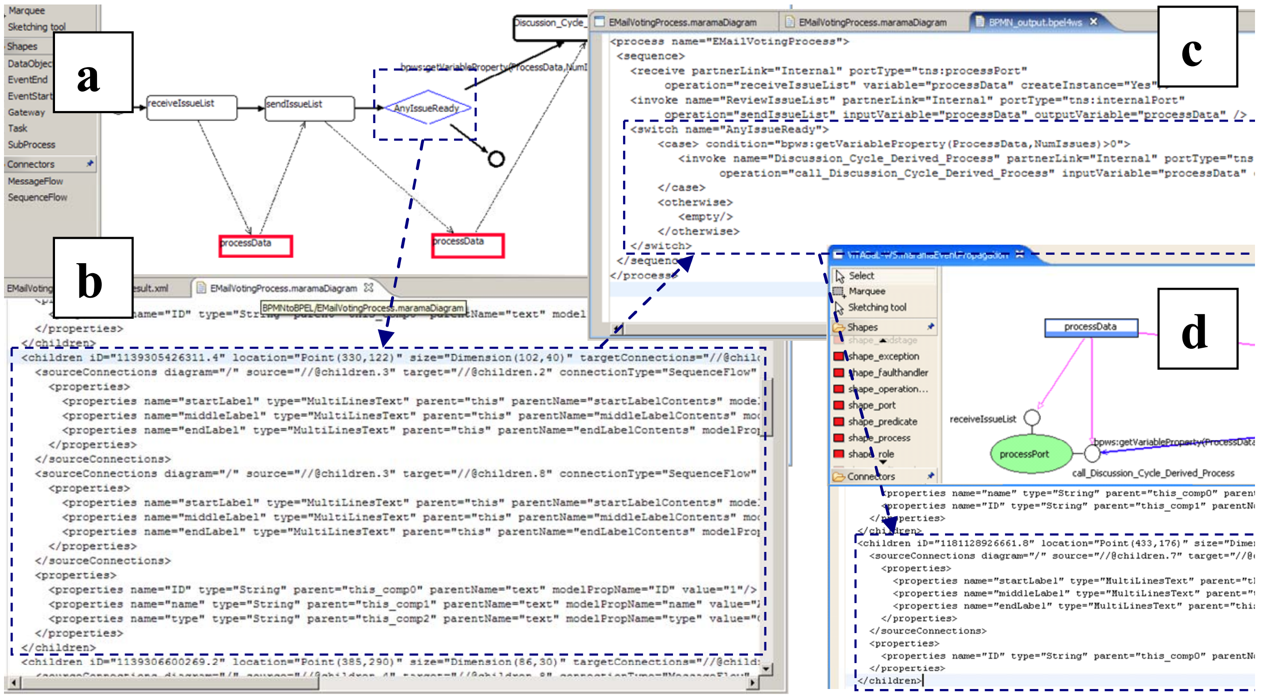
Typical approaches to building such complex data translators include: ad-hoc coding in Java, C#, or other high level languages [28]; reusing existing translators (if suitable) [16]; writing XSLT, ATL, QVT or other (semi-)declarative translation scripts [3][10][31]; using tools that generate translators from high-level, visual specifications [1][12][13], and attempting to automatically infer data mappings [5][6][8].

Using high-level specification tools that generate high quality translators is the preferred approach [1] [27][13]. This makes complex translator development faster, more scalable and maintainable, and results in higher quality translators than ad-hoc coding or reuse of less appropriate existing translators. Toolsets to generate such translators are typically either general-purpose, supporting specification of mappings between a wide range of models, or domain-specific and limited to a small range of source models and target formats. General-purpose toolsets usually provide only low-level modeling support and limited extensibility. Domain-specific translator generators provide higher-level abstractions but are often inflexible and do not support modifying built-in data mapping specifications.

We describe MaramaTorua[1], a new multi-view, semi-automated, translator specification environment which provides high-level mapping specifications for model transformation, code generation and data mapping. MaramaTorua is built and integrated with our Marama domain-specific visual language meta-tool. This allows Marama users to integrate multiple tool models, transform models and generate code and scripts using MaramaTorua. It also means the visual appearance, editing behavior and model semantics checking of MaramaTorua itself can be tailored by Marama users to support new data structures and mappings for domain-specific data mapping tasks. MaramaTorua "mapping agents" assist end users to interactively specify data mappings for large source and target schemas. A variety of target generator technologies are supported including XSLT, ATL and Java. Users can even use MaramaTorua to write new translator code generators for itself e.g. a QVT or JET data translation code generator.

We begin by presenting a motivating example for MaramaTorua, key requirements for such a data

[1] Marama is New Zealand Maori for "moon", the generator of an eclipse. Torua is Maori for "transformer".



```
<process name="EMailVotingProcess">
 <sequence>
  <receive partnerLink="Internal" portType="tns:processPort"
       operation="receiveIssueList" variable="processData" createInstance="Yes"
  <invoke name="ReviewIssueList" partnerLink="Internal" portType="tns:internalPort"
       operation="sendIssueList" inputVariable="processData" outputVariable="processData" />
  <switch name="AnyIssueReady">
      <case> condition="bpws:getVariableProperty(ProcessData,NumIssues)>0">
         <invoke name="Discussion_Cycle_Derived_Process" partnerLink="Internal" portType="tns
              operation="call_Discussion_Cycle_Derived_Process" inputVariable="processData"
      </case>
      <otherwise>
         <empty/>
      </otherwise>
  </switch>
 </sequence
</process>
```

```
     ame="ID" type="String" parent="this_comp0" parentName="text" model
  </properties>
</children>
<children iD="1139305426311.4" location="Point(330,122)" size="Dimension(102,40)" targetConnections="//@chil
  <sourceConnections diagram="/" source="//@children.3" target="//@children.2" connectionType="SequenceFlow"
    <properties>
      <properties name="startLabel" type="MultiLinesText" parent="this" parentName="startLabelContents" mode
      <properties name="middleLabel" type="MultiLinesText" parent="this" parentName="middleLabelContents" mod
      <properties name="endLabel" type="MultiLinesText" parent="this" parentName="endLabelContents" modelPro
    </properties>
  </sourceConnections>
  <sourceConnections diagram="/" source="//@children.3" target="//@children.8" connectionType="SequenceFlow"
    <properties>
      <properties name="startLabel" type="MultiLinesText" parent="this" parentName="startLabelContents" mode
      <properties name="middleLabel" type="MultiLinesText" parent="this" parentName="middleLabelContents" mod
      <properties name="endLabel" type="MultiLinesText" parent="this" parentName="endLabelContents" modelPro
    </properties>
  </sourceConnections>
  <properties>
    <properties name="ID" type="String" parent="this_comp0" parentName="text" modelPropName="ID" value="1"/>
    <properties name="name" type="String" parent="this_comp1" parentName="text" modelPropName="name" value="
    <properties name="type" type="String" parent="this_comp2" parentName="text" modelPropName="type" value="
  </properties>
</children>
<children iD="1139306600269.2" location="Point(385,290)" size="Dimension(86,30)" targetConnections="//@child
```

```
    <properties name="name" type="String" parent="this_comp0" paren
    <properties name="ID" type="String" parent="this_comp1" parentN
  </properties>
</children>
<children iD="1181128926661.8" location="Point(433,176)" size="Dime
  <sourceConnections diagram="/" source="//@children.7" target="//@
    <properties>
      <properties name="startLabel" type="MultiLinesText" parent="t
      <properties name="middleLabel" type="MultiLinesText" parent="
      <properties name="endLabel" type="MultiLinesText" parent="thi
    </properties>
  </sourceConnections>
  <properties>
    <properties name="ID" type="String" parent="this_comp0" parentN
  </properties>
</children>
```

Figure 1. Examples of complex model transformations.

mapping tool, and critiques of existing approaches. An outline of the MaramaTorua approach follows including its genesis in our earlier work. We then illustrate usage and describe architecture and implementation approaches. Our experiences with the tool to date, and an evaluation of its strengths and weaknesses follow together with a summary and directions for future research.

2. Motivation

Consider the problem of translating a software tool model from one format to another. For example, in Figure 1 (a) a Business Process Modeling Notation (BPMN) diagram, built with our Marama meta-tool, represents a simple business process. Figure 1 (b) is the XML encoding of this model. We want to translate this into a Business Process Execution Language for Web Services (BPEL4WS) specification, such as the one in Figure 1 (c) for execution on a BPEL workflow server or to another process model format for use in our ViTABaL-WS tool [21] (Figure 1 (d)). A few of the relationships between the BPMN document and target BPEL4WS or ViTABaL-WS models are also shown.

Example mapping situations that need to be supported include:

- Simple copy of the same values with different XML tags
- More complex, formula-based conversion of date or address values; or a complex repeating record structure into a non-repeating hierarchical one.
- Transformation of one Marama tool model to another to facilitate tool integration or model-driven development.
- Translation of a document from one format to another to facilitate business-to-business (B2B) enterprise system integration e.g. an ebXML document to an EDI message [8][14].
- Export of data from one CAD package to another e.g. a building design to a wiring design tool [2].
- Translation of a UML XMI model into a browser renderable form e.g. Scalable Vector Graphics (SVG), both represented by XML documents [23].

Implementing ad-hoc programming language code for each translation is time-consuming, error-prone and difficult to maintain. Even using more declarative transformation and code generation-support languages like XSLT, QVT, Velocity and JET is challenging. Ideally an IDE supporting specification of complex inter-model relationships and generation of translator implementations would be used.

Over many years of research in this domain, we and others have identified several key requirements for such data mapping modeling and translator generator tools. These include:

- Use of a variety of appropriate, high-level domain-specific visual languages for representing both models and mappings
- Ability to specify complex mappings easily
- Tool support to assist the user to manage the complexity of very large model mappings
- Fully-automatic generation of translators from high-level specifications
- Incremental support for testing and refining mapping specifications
- Integration with other common software engineering tools, particularly IDEs

Data transformation for data-oriented tool integration has usually been done on an ad-hoc basis with adaptors or translators being authored to link tools as required [14]. While architectures have been developed to simplify this process [16], these still typically are reused by writing code specific to the tool integration task at hand. A number of tool interchange "standards" have been developed to attempt to solve this, e.g. XMI (models) and GXL (graphs) are two common approaches [17][24]. However, many situations require translation between quite different tool models to facilitate data exchange [30].

B2B information exchange via XML-based formats has lead to development of a number of data mapping tools for XML messaging and document exchange. These include StylusStudio, Altova MapForce, and our own Rimu EDI/XML data mapper [1][14][27]. While such tools provide reasonable abstractions for document transformation, they are often poorly integrated with other development tools. Many lack tool extension capability to handle complex data mapping problems not directly supported in the tool's domain-specific language. Often the tools require awareness of the target generated translator implementation language and may require direct use of that language for non-trivial mapping specifications.

The rise in interest in model-driven development has led to a need to transform models of software from high-level to low-level, and eventually to code. A number of model transformation tools have been developed to support this, using textual domain-specific languages. These include QVT, ATL and Apache Velocity [3][10][31]. Some, like ATL, have relatively good IDE support via Eclipse and Visual Studio. However, data transformation authors still need to write scripts and expressions over abstract data structures, often using XPath, XQuery and similar query languages. Higher-level mapping tools have been prototyped that generate QVT and similar implementations from more declarative specifications [3][4][18][19]. However, mapping large complex data models textually means translator authors still lack support for visualizing relationships and understanding the larger structure of the mappings and schema [13].

Code generators have been an active area of research and practice for many years. Traditionally these were either custom-written code, template engines like Eclipse JET [11] or Apache Velocity [3], or "unparsers" that specified mappings from abstract syntax representations to code. As with data mapping languages like XSLT, model transformation with languages like ATL is difficult and time-consuming for large models. Recent approaches have supported code generator extension and reuse via techniques such as aspect-oriented extension [28] and composition [16]. While improving mapper development, these approaches still use textual specifications and lack high-level visualizations of the mapping process.

Attempts have been made to automatically derive model translators, particularly for automated database schema mapping [6][7][26]. These are attractive as a translator can be theoretically synthesized by comparing source and target schema automatically. In reality only limited parts of complex model transformation can be done in a purely automated way [26][30][5]. Our experience is that semi-automated mapping agents can greatly assist translator authors but must be limited to subsets of schema and coupled with good schema and mapping visualization support [5].

In our own prior work we have developed a range of data mapping specification and translator tools. These include a declarative data mapping language and interpreter for CAD tool integration; an EDI message mapping specification and domain-specific language generator tool; a business data mapping tool; and mapping tool supporting notation transformation for CASE tool integration [5][13][20]. Empirical studies show the tools provide good support for specifying and generating complex data mappings in their respective domains. However they suffer from similar limitations:

- a single, one-size-fits-all domain-specific visual language;
- limited ability to generate alternate translator codings;
- no integration with other software tools;
- no support for managing large source and target data models;
- minimal ability for end-user customization

Each of these problems, model transformation, code generation and tool data transformation, is actually a subset of the general data mapping problem. In each case we are trying to translate a model in one format to another, either: an orthogonal model (for tool import/export and integration); a lower-level model (for code generation and model-driven engineering); or a higher-level, more abstract model (for reverse-engineering and data and/or software visualization). All benefit from the provision of high-level, visual data mapping support; assistance to the user for large data mapping problems; automatic data mapping generation from the high-level mapping language; and an integrated IDE with other software tools.

3. Our approach

We have looked to address the limitations of our own and other data mapping generation tools via a multi-view IDE extension approach. Our experiences have shown us that a single visual metaphor for source and target data models and mappings is insufficient [13] and the mapping specification tool must be well integrated with other software tools. Thus for Marama Torua, we support multiple views of the data models and mapping specifications using different visual metaphors and have implemented the tool as an Eclipse plug-in enabling close data, control and presentation integration with a wide range of other software tools. The MaramaTorua meta-model enables a wide range of source and target data models and complex mapping specifications to be represented. This allows the tool to be applied widely; provides users with a wide range of visual metaphors; and allows multiple target translator implementation code generators to be supported.

Figure 2 outlines MaramaTorua's use to engineer complex data mapping translators. The user imports existing XML Schema to provide the source and/or target data format specifications (1). These schema may be manually created, automatically generated e.g. from the Marama meta-tool, or 3rd party. If no schema exists (a common problem) users may define one with MaramaTorua's schema editor or an existing Eclipse schema editor (2). If the user has example XML data models they may ask for a schema to be generated using a web services link to the Microsoft schema inference engine (3). The user then specifies mappings between source and target schema elements (4). These may be quite simple e.g. copy source to target data item, or complex e.g. iterate over source collection filtering on specified data item values and create new target data structures. The source and target schema may be large e.g. >1,000 elements in which case the user is assisted by "mapping agents" that provide interactive suggestions based on source/target element tag names equivalence or similarity; element types; complex structure similarities; and example data item equivalences. The user may reuse mapping functions from MaramaTorua's extensible library. Multiple views allow the user to split up complex mappings. In addition, multiple *representations* of the source and target schema are supported, including a tree-based schema view type and a "business form" view type.

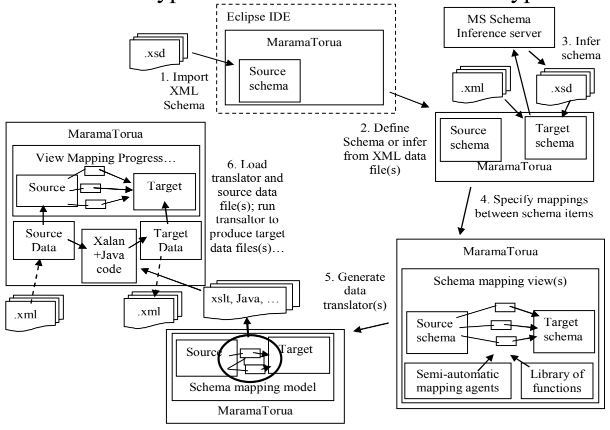


Figure 2. Outline of using MaramaTorua.

After completing the mapping specification the user requests generation of a translator. MaramaTorua uses mapping specifications and reusable mapping functions to synthesize a translator implementation. Typically this is a set of XSLT scripts with extra Java classes for e.g. string manipulations (which are poorly supported by XSLT) (5). Other target translator implementation languages can also be used e.g. ATL or pure Java code. Users can test their translators by loading the translator code and example source data files into MaramaTorua. The user can view the result of running the translator on all or parts of the example input models (6). MaramaTorua is integrated with the Marama meta-tool and its generated translators can be used directly within Marama-generated tools to support model integration, translation, and code generation.

4. Example input

To demonstrate MaramaTorua's capabilities we revisit the examples in Figure 1. Here the tool developer/ integrator wants to specify a code generator (BPMN->BPEL4WS) and an import/export mechanism (BPMN<->ViTABaL-WS) for their BPMN tool.

MaramaTorua provides two additional meta-tool specification views for the Marama meta-tool: a schema specification view and a data mapping view. A tool developer first obtains an XML Schema for BPMN. In this case, they import the meta-model for a Marama-specified BPMN tool into a new Marama Torua schema view. Figure 3 (a) shows part of the Marama meta-model view for the MaramaBPMN tool. Figure 3 (b) shows this schema represented in a MaramaTorua schema view. Marama uses a simple

Extended Entity-Relationship meta-model notation. MaramaTorua uses a more abstract representation than either EER or XMLSchema to minimize unnecessary detail and to support different underlying schema models than just XMLSchema, e.g. EXPRESS-G.

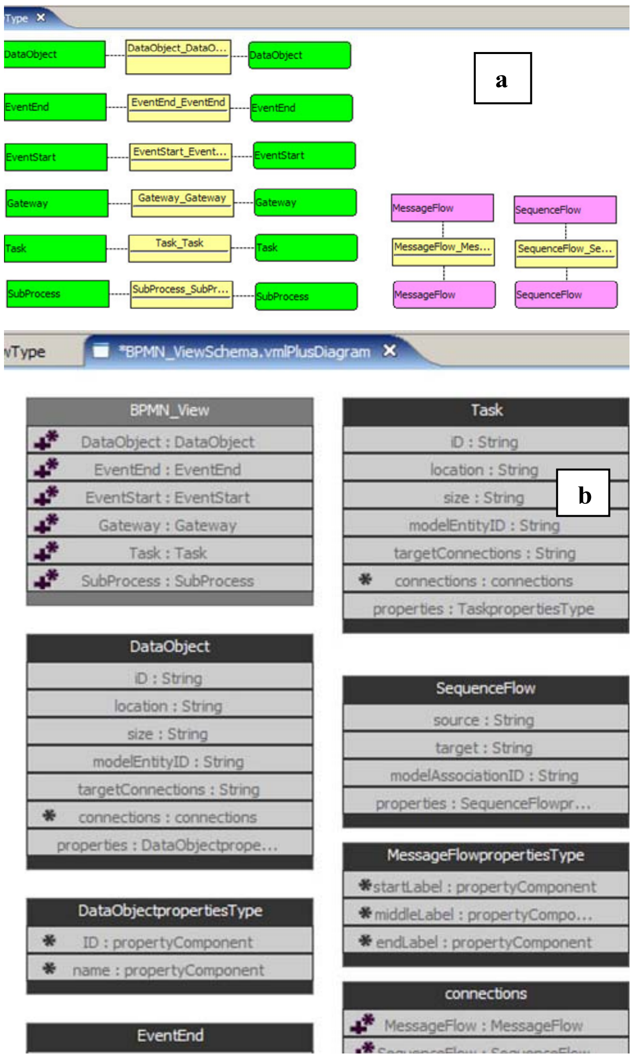


Figure 3. Viewing an XML Schema for BPMN imported from the Marama meta-modelling tool.

To map BPMN to ViTABaL-WS and BPEL4WS, the user must also obtain or define their schema. The ViTABaL-WS schema is also imported from its Marama meta-model but we lack one for BPEL4WS. We could manually define one with the schema editor, but this would be time consuming. However, we have an example BPEL4WS XML data file and can obtain an initial schema for it by submission to the Microsoft schema inference web service. This is done via a MaramaTorua dialog box and the inferred schema is loaded into a MaramaTorua schema editor view.

The tool developer now creates two mapping specifications, to respectively map MaramaBPMN data to ViTABaL-WS and BPEL4WS data. Both use moderately complex mapping conditions, repeating group copies and formulae. In Figure 4 (a) the user begins to specify data mappings between BPMN schema items (left) and BPEL4WS schema items (right). Here the user begins by expanding the root schema nodes for the BPMN (left) and BPEL4WS models (right). The user specifies mappings between elements in the source and target models via drag and drop. Mappings can be one-to-one, one-to-many, many-to-one or many-to-many. A central mapping node captures the relationship. The tool developer uses this to specify calculations or functions needed to complete the transformation. In Figure 4 (a) a simple one-to-one equality mapping has been specified between the *BPMN name element* and *BPEL4WS name element*. The mapping of *BPMN eventStart* to *BPEL4WS process switch* is more complex so has been expanded as sub-mapping (eventStartMapping) immediately below.

As the mapping specification develops, challenges arise. Some transformations are conditional on content i.e. not dependent on the static model but on values the XML instance data will contain. For example in Figure 4 (b), the mapping of the *BPMN Process process* element to the *BPEL4WS process sequence invoke* element is conditional (represented by the rhombus) on whether its *id* value equals the value of the decimal *id* parameter passed from a higher level mapping, and also on not being a *receive process type*. Figure 4 (c) shows the conditional expression involved specified using a structured formula builder. This has XPath-style expressions to access the *id* and *type* element values. More complex paths can be easily specified.

Mappings that are more complex than equalities are specified using mapping formulae. These may involve parameterized mapping functions (predefined or user specified). For example in Figure 4 (d), a parameterized substring function maps the *BPMN Process process* name to the *BPEL4WS process sequence invoke* element's name.

For complex mappings, MaramaTorua integrates AXSM, an automated mapping suggestion tool [5] into the mapping view so the user does not have to manually specify every mapping relation. This feature heuristically predicts and visualizes potential mappings to the end user, who may then choose to accept or decline any of the mapping suggestions; resulting in further cycles of automated prediction and updated

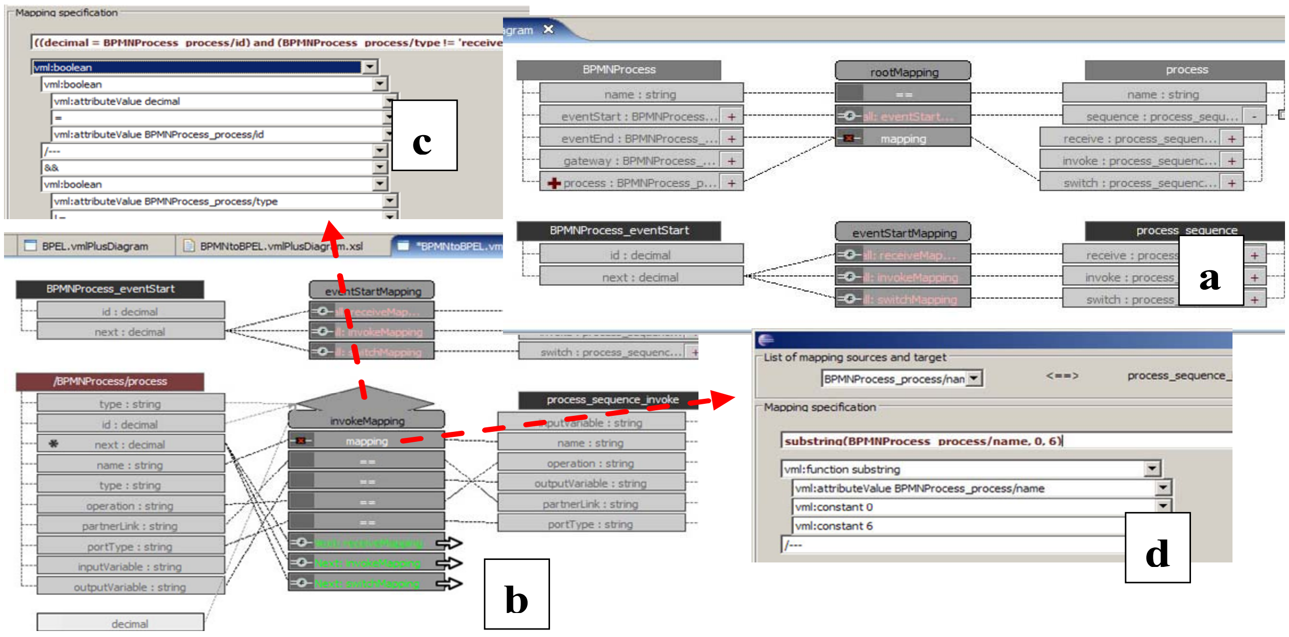


Figure 4. Specifying mapping relationships in the mapping view.

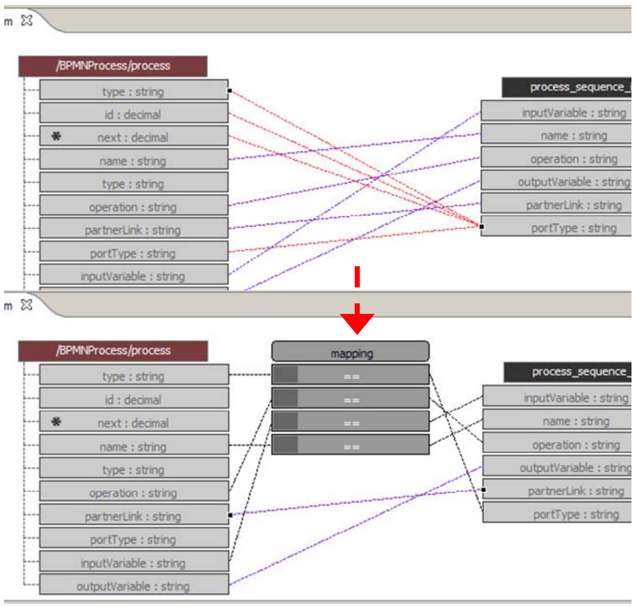


Figure 5. Semi-automated data mapping.

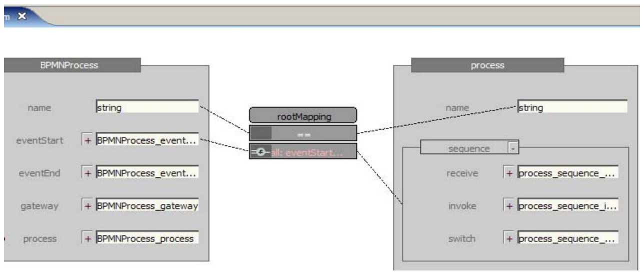


Figure 6. Using the form-based mapping view.

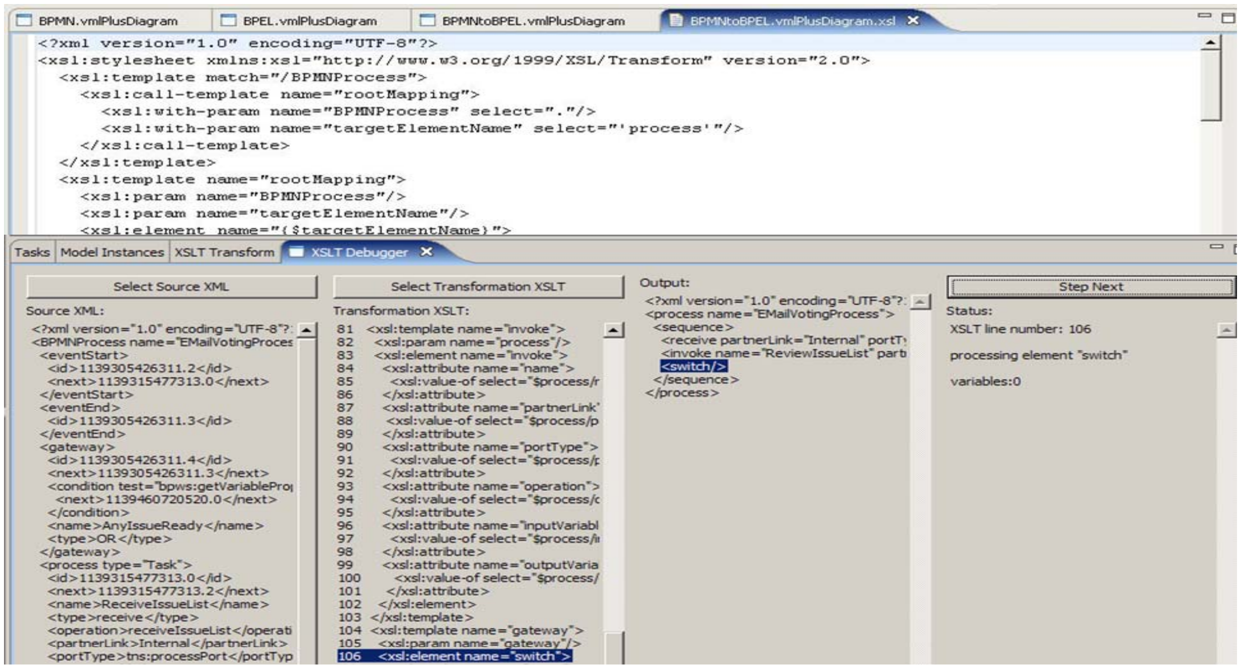
mapping suggestions based on prior user selections. AXSM uses extensible mapping agents written in Java. MaramaTorua integrates the AXSM extensible library into its run-time environment so new algorithms for mapping agents may be added dynamically within the IDE on the fly. Figure 5 (top) shows an example of mapping predictions. In several cases unique suggestions are made, in others (such as to the *BPEL4WS portType* attribute) multiple suggestions are made which the user must select between. In Figure 5 (bottom) the user has accepted several suggestions (including one of those for portType) and these have been converted into equality mappings. Two suggested mappings (bottom) await user acceptance or rejection.

As an alternative to the "conventional" indented hierarchical view of the mapped schemas, Marama Torua supports a form based mapping metaphor we have previously developed [13]. Here the schema may be structured in a similar fashion to a conventional business form, with structure depicted by containment and field placement rather than hierarchy. Mappings are specified by drag and drop between form fields. This metaphor is not particularly apt for our running example, which is programmer centric. Nevertheless, Figure 6 shows the high level BPMNProcess to BPEL4WS process mapping represented using this metaphor. For non programmers, e.g. business analysts, this metaphor offers a more user friendly representation than the conventional tree based representation. We are also exploring other types of concrete metaphor.

The next step is to generate mapping code and test it. Figure 7 shows this process. At top is a fragment of the XSLT code generated for the BPMN to BPEL4WS mapping. Below is the invocation panel used to execute and test mappings. At left, an XML source file is shown. In the next sub-panel, the generated XSLT mapping is shown. The third panel shows XML output generated as the mapping is executed. At right a debugger interface allows step by step execution of the mapping to be undertaken with status information such as the current XSLT line number and element shown. Corresponding elements in the XSLT and XML code are also highlighted.

As the final step the generated mapper is installed in the Marama BPMN tool, so its users can generate BPEL from their BPMN diagrams. At left of Figure 8, the user selects the MaramaTorua generated code to add to the BPMN tool. This is added as a plugin, attached to a newly defined *handler* (a behavioural extension to the Marama tool), as shown in the next screen dump. This handler is invoked from a context sensitive menu element in the BPMN diagram editor view and uses the BPMN tool model as its input. The resulting BPEL output is shown at the right.

MaramaTorua, being implemented using Marama, has highly customizable appearance and functionality. Users may customize its icons and connectors and add their own event handlers and menu actions by writing extensions as Marama Java event handlers. Useful example extensions include: visualising the mapping in a new display format; visualization layout algorithms; and new interactions with user developed tools.



```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:template match="/BPMNProcess">
    <xsl:call-template name="rootMapping">
      <xsl:with-param name="BPMNProcess" select="."/>
      <xsl:with-param name="targetElementName" select="'process'"/>
    </xsl:call-template>
  </xsl:template>
  <xsl:template name="rootMapping">
    <xsl:param name="BPMNProcess"/>
    <xsl:param name="targetElementName"/>
    <xsl:element name="{$targetElementName}">
```

Source XML:

```
<?xml version="1.0" encoding="UTF-8"?:
<BPMNProcess name="EMailVotingProces
 <eventStart>
  <id>1139305426311.2</id>
  <next>1139315477313.0</next>
 </eventStart>
 <eventEnd>
  <id>1139305426311.3</id>
 </eventEnd>
 <gateway>
  <id>1139305426311.4</id>
  <next>1139305426311.3</next>
  <condition test="bpws:getVariablePro
   <next>1139460720520.0</next>
  </condition>
  <name>AnyIssueReady</name>
  <type>OR</type>
 </gateway>
 <process type="Task">
  <id>1139315477313.0</id>
  <next>1139315477313.2</next>
  <name>ReceiveIssueList</name>
  <type>receive</type>
  <operation>receiveIssueList</operati
  <partnerLink>Internal</partnerLink>
  <portType>tns:processPort</portTyp
```

Transformation XSLT:

```
81   <xsl:template name="invoke">
82     <xsl:param name="process"/>
83     <xsl:element name="invoke">
84       <xsl:attribute name="name">
85         <xsl:value-of select="$process/r
86       </xsl:attribute>
87       <xsl:attribute name="partnerLink"
88        <xsl:value-of select="$process/p
89       </xsl:attribute>
90       <xsl:attribute name="portType">
91         <xsl:value-of select="$process/p
92       </xsl:attribute>
93       <xsl:attribute name="operation">
94         <xsl:value-of select="$process/c
95       </xsl:attribute>
96       <xsl:attribute name="inputVariabl
97         <xsl:value-of select="$process/i
98       </xsl:attribute>
99       <xsl:attribute name="outputVaria
100        <xsl:value-of select="$process/
101      </xsl:attribute>
102    </xsl:element>
103  </xsl:template>
104  <xsl:template name="gateway">
105    <xsl:param name="gateway"/>
106    <xsl:element name="switch">
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?:
<process name="EMailVotingProcess">
 <sequence>
  <receive partnerLink="Internal" portTy
  <invoke name="ReviewIssueList" parti
  <switch/>
 </sequence>
</process>
```

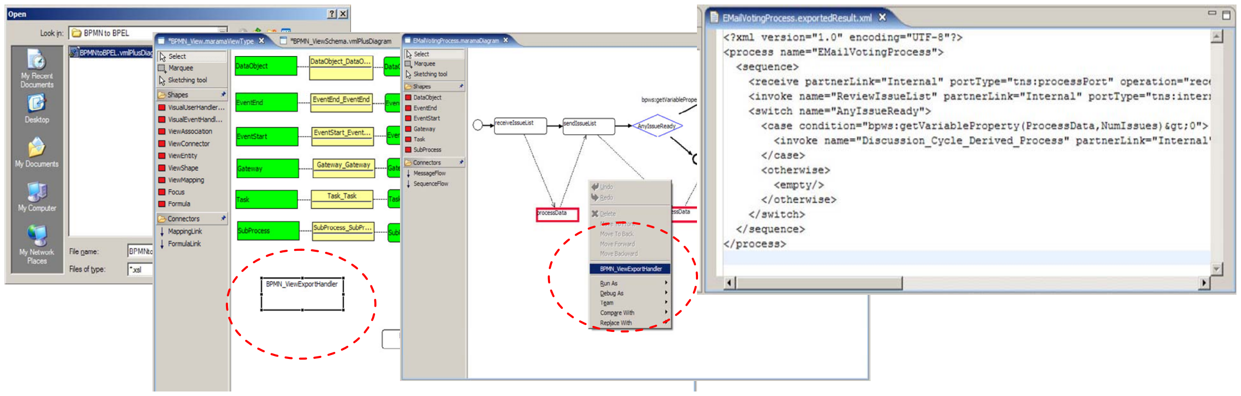
Status:

```
XSLT line number: 106

processing element "switch"

variables:0
```

Figure 7. Generating and testing translators.



```
<?xml version="1.0" encoding="UTF-8"?>
<process name="EMailVotingProcess">
  <sequence>
    <receive partnerLink="Internal" portType="tns:processPort" operation="rec
    <invoke name="ReviewIssueList" partnerLink="Internal" portType="tns:inter
    <switch name="AnyIssueReady">
      <case condition="bpws:getVariableProperty(ProcessData,NumIssues)&gt;0">
        <invoke name="Discussion_Cycle_Derived_Process" partnerLink="Internal'
      </case>
      <otherwise>
        <empty/>
      </otherwise>
    </switch>
  </sequence>
</process>
```

Figure 8. Installation of a mapper into a Marama-generated visual modeling tool.

5. Architecture/implementation

We built MaramaTorua with our Marama meta-tool [15]. We have integrated MaramaTorua back into Marama as a meta-tool component to support specification of tool integration, model translation and code generation for Marama-based tools. However, we have maintained separation of Marama and Marama Torua, both being fully functional as stand alone tools.

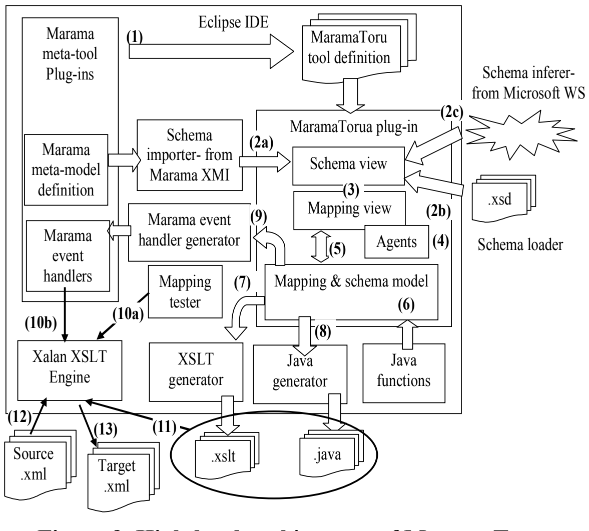


Figure 9. High-level architecture of MaramaTorua.

Figure 9 is the high-level architecture of Marama Torua. It is defined using Marama and instantiated as models and editors in Eclipse (1). Users import source and target schema into the tool from Marama meta-model definitions (2a); XML Schema (2b); or via a web service link to the Microsoft schema inference engine to generate schema from sample XML data files (2c). Users may modify inferred schema or create new ones from scratch (3). A set of "mapping agents" assist with identifying mappngs between large schemas (4). As schema and data mapping views are manipulated a MaramaTorua model is constructed (5). The user may make use of pre-built functions from a Java (or other translator implementation e.g. XPath for XSLT) library for very complex transformations (6). If necessary, extra functions, e.g. for date conversions, text and address parsing, can be implemented using Eclipse and exposed via the MaramaTorua library.

The completed mapping is then used to generate the translator, e.g. as XSLT (7), helper functions/ scripts, e.g. Java functions for text parsing (8), and possibly a Marama event handlers to be plugged into a Marama tool (9) so the tool can invoke the translator. The generated data mapper can be tested in Marama Torua (10a) or from within a Marama-generated tool (10b). To use a translator, its implementation is loaded and run by an appropriate engine e.g. Xalan for XSLT. The data mapper is given one or more source XML files and it transforms these according to its specification into one or more target XML files

In developing MaramaTorua several key design and implementation decisions were made. Firstly, the visual notations used are abstract and do not adhere to an existing language. They were initially developed from our experience with data mapping problems in the health messaging and construction tool integration domains, areas where data mapping problems are common. The visual notations declaratively highlight the structural differences between the schemas being mapped instead of visually expressing how mapping should be undertaken procedurally. This has advantages of clarity and readability, where details such as individual formulae (best expressed textually anyway) are not shown. The approach also provides the ability to easily extend the tool to generate additional output formats. Secondly we made a clean separation of the meta-tool from MaramaTorua. Both plug-ins are fully functional as stand alone tools. The integration of the two was undertaken at a higher level. This coordinates the flow of the data between the two plug-ins by passing meta-tool definitions as mapping schemas and linking generated mapping outputs with tools designed by the meta-tool. Finally, MaramaTorua takes advantage of modularized components, to ease the development costs of writing new schema loaders, output generators, and mapping agents. Loading schemas can be undertaken in many ways, including; loading from XSD files with schema loaders; loading from an XML instance by creating a schema through Microsoft XSD inference tool; and importing from the meta-tool. New schema loaders can be written giving MaramaTorua the extensibility to load schemas from any possible data source or data structure. Since the loaded schemas are converted to abstract MaramaTorua notations, mappings can be specified between schemas that have been generated from different source formats. The same is true for writing output generators. There is a high level of customizability here with output generation to any form of language possible.

6. Discussion

We have used MaramaTorua on a wide range of mapping problems including:

- BPMN->BPEL4WS code generation;
- generation of Java from a Marama UML tool;
- import of XMI design models into a UML tool;
- conversion of Marama diagrams into SVG format for thin-client web browser rendering;

- data exchange between the Web of Patterns (RDF) and the MaramaDPTool design pattern tool [9];
- several large XML to XML data translations for E-business application integration.

Some of these are complex data translations; e.g. the E-business models have over 1,000 XML Schema elements. MaramaTorua allows users to not only define these complex data mappings but do so incrementally and with incremental testing. We have further evaluated the effectiveness of MaramaTorua in two ways: an end user evaluation; and its completeness against the key requirements in Section 2.

Our end user evaluation had four experienced data translator implementers carry out a set of mapping tasks (parts of the BPMN->BPEL4WS problem). They used MaramaTorua to model the schema, specify a range of mappings, generate an XSLT-based translator, and test it. Overall results were very favorable with all users able to carry out the task in orders less time than directly implementing Java or XSLT translators. Users expressed overall satisfaction with MaramaTorua's capabilities. Some difficulties found were modeling conditional mappings and string parsing operations and the specification of complex expressions using the MaramaTorua formula editor. Users desired a "design by example" approach for the latter using actual source and target values with the tool inferring conversions.

From Section 2's requirements, MaramaTorua is a highly integrated environment for modeling and generating complex data mapping implementations using a set of high level domain specific visual languages and both a tree-based, schema-oriented visual language and a "business form" based visual language for modeling mappings. Users can import source and target schema from existing XML Schema, the Marama meta-tool, model a schema themselves, or have one inferred from example data. MaramaTorua supports specification of complex large-scale data mapping using multiple views, elision, and mapping agents to help manage very large search spaces. It also provides a testing environment allowing users to incrementally specify and test mappings. In addition, MaramaTorua provides close integration with Marama allowing tool designers to use MaramaTorua as another meta-tool capability, generating event handlers that plug into Marama-generated tools to support import/ export, model transformation and code generation.

Limitations involve specification of complex expressions and collection-manipulating operations (difficult in most mapping specification tools) and the lack of use of "concrete" example data during the data mapping process. We have previously added example data to aid mapping specification in earlier tools [14][20] and plan to use a similar approach for MaramaTorua. We are developing structured textual representations of source and target schema using Eclipse's textual editor framework to support a metaphor like the form-based mapping views where source and/or target schema have a textual concrete representation e.g. code or scripts.

A more fundamental limitation is that to be able to specify mappings that exceed a certain complexity level, the user needs to be sufficiently familiar with the target language (XSLT) to be able to specify the mapping textually in the first place. This means that when the problem is simple, MaramaTorua can be used effectively to aid people who have little knowledge of XSLT to specify mappings, but when the problem is complex, the tool is more of a visualization aid, with lower levels of productivity enhancement. This could be improved by introducing more abstract notations tailored to common structural mapping problems. We should stress that this point relates to the mapping's structural complexity not its *size*. The latter is well addressed using the mapping agents which provide a high level of productivity enhancement.

Areas for future work include the following. Providing additional, more "concrete" views would better support mapping specification. These could include code/script text views for code generation and shape/icon specification for mapping from one diagram format to another. It is also desirable to support example data values in-situ in the source/target schema elements. The loaded values can be used to guide manual specification of mappings by the designer or as additional mapping suggestions for the semi-automatic mapper. At execution time, this provides a natural mechanism for visualizing execution behavior. The system does not currently support translation of constraints, which are essential for full model transformation. We are currently developing a higher level model transformation specification language that addresses this shortcoming.

7. Summary

Implementing data-based integration, import-export, model transformation and code generation capabilities in software tools is challenging. We have developed an integrated, visual language-based toolset, MaramaTorua, to support these activities for Eclipse-based software tools. Users import, define or have inferred XML Schema which they specify mappings between, with the aid of mapping agents for large-scale problems. Data translators are synthesized from these schema mapping specifications, including XSLT and Java-based implementations. MaramaTorua provides an integrated environment for modeling, generating and testing these data mappers. These generated data mappers may be seamlessly integrated into other Marama-generated tools to support their import/export, model transformation and code generation needs.

8. References

[1] Altova, MapForce , www.altova.com/products/mapforce/data_mapping.html.

[2] Amor, R., Hosking, J.G., Mugridge, W.B. ICAtect-II: a framework for the integration of building design tools, In *Automation in Construction*, 8(3) 1999, 277-289.

[3] Apache Velocity, http://velocity.apache.org/

[4] Bichler, L. A flexible code generator for MOF-based modeling languages, *Proc 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*.

[5] Bossung, S., Stoeckle, H., Grundy, J.C., Amor, R. and Hosking, J.G. Automated Data Mapping Specification via Schema Heuristics and User Interaction, In *Proc Int Conf on Automated Software Engineering*, Linz, Austria, September 20-24, IEEE CS Press, pp. 208-217.

[6] Bottcher, S. and Grope, S. Automated data mapping for cross enterprise data integration, In *Proc 2003 International Conference on Enterprise Information Systems*, 2003.

[7] Bourret, R., Bornhövd, C., Buchmann, A.P.: A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases, In *Proc 2nd Intl Workshop on Advanced Issues of Electronic Commerce and Web-based Inf Systems*, San Jose, California, June, 2000.

[8] Damm, D., Hakimpour, F. and Geppert, A. Translating and Searching Service Descriptions Using Ontologies, LNCS, Volume 2681/2003, Springer.

[9] Dietrich, J., Elgar, C.: Towards a Web of Patterns, Workshop on Semantic Web Enabled Software Engineering (SWESE) Proc ISWC 2005, Galway, Ireland, 2005.

[10] EclipseModel-to-model transformations, www.eclipse.org/m2m/

[11] Eclipse Model to Text, http://www.eclipse.org/modeling/m2t

[12] Goulde, M.A. Microsoft's BizTalk Framework adds messaging to XML, *E-Business Strategies & Solutions*, Sept. 1999, 10-14.

[13] Grundy, J.C, Hosking, J.G., Amor, R., Mugridge, W.B., Li, M. Domain-specific visual languages for specifying and generating data mapping systems, *JVLC*, vol. 15, no. 3-4, June-August 2004, Elsevier, pp 243-263.

[14] Grundy, J.C., Mugridge, W.B., Hosking, J.G. and Kendal, P. Generating EDI Message Translations from Visual Specifications, In *Proc Automated Software Engineering Conference*, San Diego, 2001, IEEE, pp. 35-42.

[15] Grundy, J.C., Hosking, J.G., Zhu, N. and Liu, N. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications, In *Proc Automated Software Engineering Conf*, Tokyo, 24-28 Sept 2006, IEEE.

[16] Henthorne, C. and Tilevich, E. Code Generation on Steroids: Enhancing COTS Code Generators via Generative Aspects, In *Proc 2nd International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques*, Minneapolis, 2007, IEEE CS Press.

[17] Holt, R.C., Winter, A., and Schurr, A., GXL: Toward a Standard Exchange Format, In *Proc Seventh Working Conference on Reverse Engineering*, IEEE CS Press, 2000.

[18] Lengyel L., Levendovszky T., Mezei G. and Charaf H., Model-Based Development with Strictly Controlled Model Transformation, *Proc 2nd Intnl Workshop on Model-Driven Enterprise Information Systems*, Paphos, Cyprus, 2006.

[19] Levendovszky T., Lengyel L., Mezei G. and Charaf H., A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, Int Workshop on Graph-Based Tools, In *Electronic Notes in Theoretical Computer Science*, 2007.

[20] Li, Y., Grundy, J.C., Amor, R. and Hosking, J.G. A data mapping specification environment using a concrete business form-based metaphor, In *Proc Int Conf on Human-Centric Computing,* IEEE, 2003 pp. 158-167.

[21] Liu, A., Grundy, J.C. and Hosking, J.G., A visual language and environment for composing web services, In *Proc Int Conf on Automated Software Engineering*, Long Beach, California, Nov 7-11 2005, IEEE Press, pp. 321-324.

[22] Liu, A., Hosking J.G., and Grundy, J.C., MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism, *Proc IEEE VLHCC*, Coeur d'Alène, Idaho, September 2007

[23] Mansfield, P. Common graphical object models and how to translate them to SVG, *SVG Open / Carto.net Developers Conference*, Zurich, July 15-17, 2002.

[24] OMG, MOF 2.0 / XMI Mapping Specification, v2.1, http://www.omg.org/technology/documents/formal/xmi.htm

[25] Pervasive.com, Pervasive Integration Manager, http://www.pervasive.com/.

[26] Rahm, E. and Bernstein, P. A survey of approaches to automatic schema matching, *JVLB*, 10(4), 2001, 334 – 350

[27] StylusStudio, StylusStudio XML-to-XML Mapper, www.stylusstudio.com/xml_to_xml_mapper.html.

[28] Swint, G. et al, GXX – Extensible, Flexible, Modular Code Generator, *Proc Int Conf on Automated Software Engineering,* Long Beach, USA, November 7-11, 2005.

[29] Tolvanen, J.-P., Making model-based code generation work - Practical examples (Part 2), *Embedded Systems Europe*, Vol. 9, 64 (March), 2005.

[30] Tratt, L., Model transformations and tool integration, *Software and Systems Modeling,* 4(2), Springer, pp. 112-122.

[31] UML-QVT, http://umt-qvt.sourceforge.net/