Copyright 2003 IEEE. Published in the Proceedings of 2003 IEEE Human-Centric Computing Conference, Auckland, New Zealand, October 2003. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 732-562-3966.

Approaches to Supporting Software Visual Notation Exchange

Hermann Stoeckle[1], John Grundy[1, 2] and John Hosking[1]

Department of Computer Science[1] and Department of Electrical and Electronic Engineering[2],
University of Auckland, Private Bag 920, Auckland, New Zealand
{herm, john-g, john}@cs.auckland.ac.nz

Abstract

A wide range of software tools provide software engineers with different views (static and dynamic) of software systems. Much recent work has focused on software information model exchange. However, most software tools lack support for exchange of information about visualisation notations (both definitions of notations and instances of them). Some basic converters have been developed to support the exchange of notation information between software tools but almost all are custom-built to support specific notations and difficult to maintain. We describe the development of several notation exchange converters for tools supporting software architecture notations. This leads to a unified converter generator framework for notation exchange.

Keywords: Static and dynamic software visualization, notation converter, converter generation

1. Introduction

A vast (and still increasing) number of software visualisation notations exist. These include static visualisation notations at varying levels of abstraction e.g. class diagrams, component diagrams and deployment diagrams [2, 15]; dependency graphs [16, 17], software architecture structure [4, 7, 17], and for dynamic visualisation e.g. call-graphs and control flow [1, 12, 16], message sequencing [17, 14, 18, 2]; dynamic architectures [6, 14]; and various run-time software characteristics like performance and resource utilisation [6, 16]. Many tools support variants of these visualisation notations and developers would like to exchange notation information between the tools from time to time e.g. exchange a UML diagram from a CASE tool to MS Visio™ for further enhancement; exchange diagram notation descriptions between CASE tools so the tools allow viewing and possibly editing of the same format notation; and exchange notation instance information with other tools to support viewing the information in a different platform e.g. web interface, 3D virtual reality interface to the visualisations etc. In addition to exchanging notation information in custom tool formats we may wish to convert between low-level display formats e.g. a CASE tool diagram into SVG, VRML or GXL exchange formats [3, 10, 11]. Currently to support these kinds of software visualisation notation exchange a custom converter or translator must be developed [9, 8, 11]. These take considerable effort to build, test and deploy and are difficult to modify if the notation itself or the tool notation information formats change. They also typically lose parts of the information in one tool when translating to another tool's notation information model.

We describe our work building several translators for different software visualisation notation formats. The original notations are those defined and used by our Pounamu meta-tool to describe various visualisation notation shapes, their properties and their relations. We have hand-built several notation converters supporting the translation of static and dynamic software architecture notation information between Pounamu and an earlier software architecture modelling tool (SoftArch), and to graph-based formats (GXL) and graphic-based formats (SVG, VRML), used by other rendering and editing tools. From insights during this work, we describe our new approach to generating notation converters from inter-visual notation mapping specifications.

2. Motivation

We have been developing Pounamu, a new meta-tool to support the specification of software engineering tools. This allows software engineers to define new meta-models and meta-views for software tools and realise tools based on these specifications. Pounamu was designed to provide thick client CASE tools to support the developer in different parts of software development. Pounamu views consist of a wide range of graphical shapes and connectors representing information about a software system, as well as dialogue-based views. Pounamu currently uses its own proprietary representation format for its software notations.

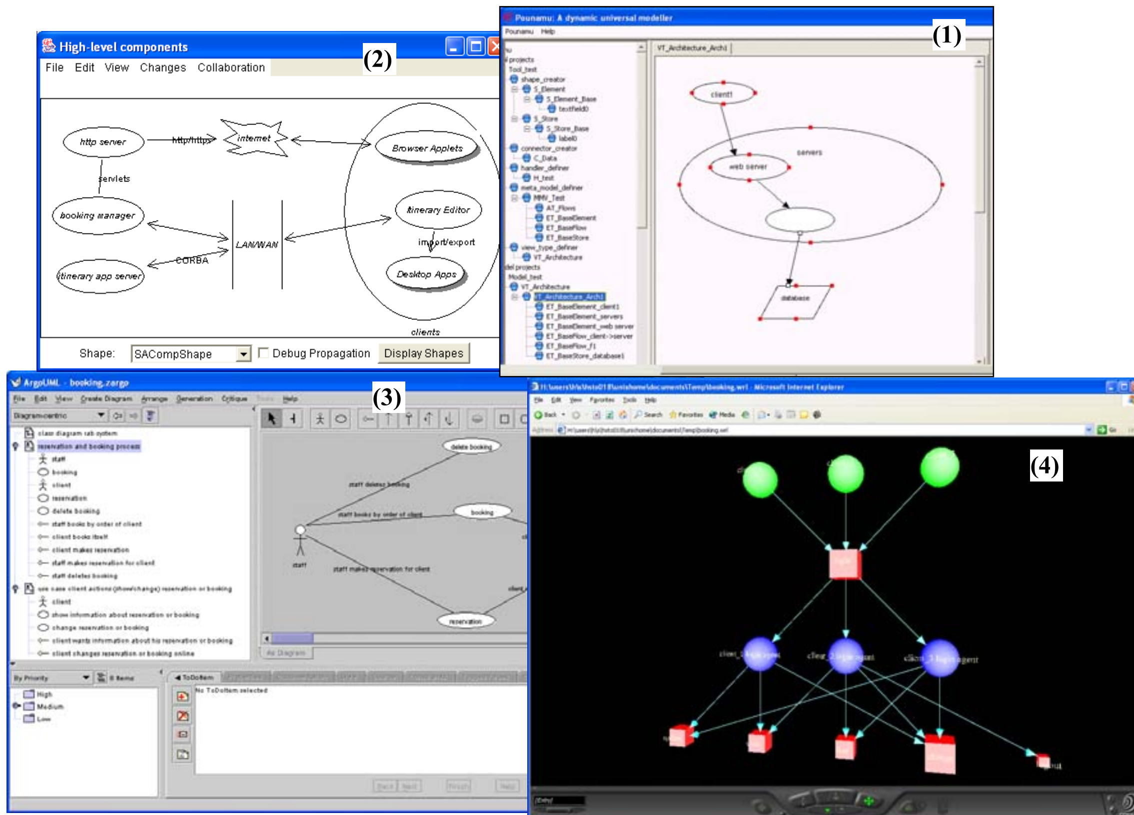

Figure 1: Some examples of software architecture visualization notation exchange between software tools.

We wish to support import and export of software visualisations (both static and dynamic) between Pounamu-based environments and other software engineering tools. We also want to allow users of Pounamu to interact with views using different viewing and editing technologies, including web-based thin client and 3D virtual reality. For complex dynamic software visualisations in particular, we want to make use of 3rd party viewing tools like 3D viewers and information visualisation tools. To achieve this, Pounamu has to support import and export of a wide range of external notation formats. This can include formats of applications dealing with graphs, view layouts, software information (e.g., UML-based models), data visualization tool formats or even translating events within a CASE tool into rule sets using Programming by Example techniques.

An example of the kinds of software visualisation notation exchange we require is shown in Figure 1. A Pounamu software architecture view (1) has exchanged different views with three other viewing tools: the SoftArch CASE tool (2); the ArgoUML CASE tool (3); and a VRML 3D visualisation of dynamic architecture performance measures (4). A range of translators that support these different kinds of notation information exchange have been used to export (and some can import) Pounamu view notation information.

3. Our Approach

We have approached this research in two phases: in the first we have hand-built several notation converter tools to enable exchange of software architecture information between software tools. From our experiences with this work we have designed a notation converter generator framework. This includes an inter-notation mapping specification language and converter generator. We are currently developing a prototype of this converter generator. Figure 2 provides a high level overview of the types of notation converters we have hand developed. These convert between the XML-based format of Pounamu and other tools' formats. Pounamu can import and export information at two levels: view-based (1), exchanging information about notations and notation instances, or model-based (2), exchanging software model information (schema or instance data). The second approach is suitable if each tool defines its own visualisation notation conventions and fully-generates its own views from the model information. The disadvantage is that there is no way to exchange information about appearance, layout and composition of visualisations.

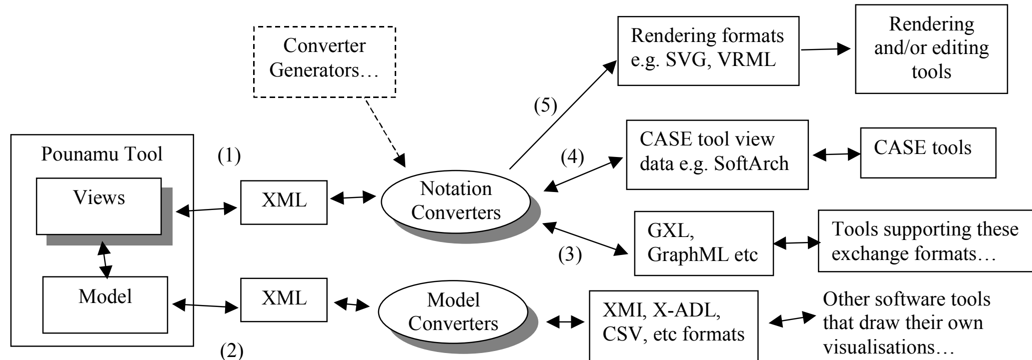

Figure 2: High-level architecture of our approach.

We use notation converters to translate Pounamu view XML save formats and (3) generic visualisation notations like GXL and GraphML, (4) tool-specific formats e.g. SoftArch's view representation or (5) low-level rendering formats e.g. SVG and VRML. Currently tool developers develop custom converters using ad-hoc architectures and implementations. Ideally we want to generate converters from specifications of mappings between notations.

4. Example Notation Converters

Here we present our experiences developing hand-coded software visualization notation converters. In these examples we use simple static and dynamic software architecture visualization notations to illustrate the kinds of notation exchanges that we want to support between software visualization tools. As a common example to illustrate these we use a software architecture description for a video store library providing different search interfaces for customers and staff. This architecture can be viewed statically and dynamically in various ways [5, 7].

4.1 Pounamu Notation to/from GXL Converter

GXL (Graph eXchange Language) is a simple XML-based graph exchange format based on relationships between nodes and edges [10]. Originally it was used for reverse engineering, where it is supported by a variety of graph-based software tools, including converters, visualization tools, graph analysis and transformation tools, and source code extractors [3, 11, 13, 21].

We have built import and export converters for moving views to/from Pounamu and GXL. Any Pounamu-designed view format can be converted to GXL or imported from GXL into Pounamu's XML-based view format description using these converters. These converters are implemented using the XSLT transformation scripting language, which converts the Pounamu view XML format to and from the GXL format. We chose XSLT to implement these GXL converters as both notations use XML-based formats for visualisation notation information description and for the ease with which we could change these XSLT scripts during converter development and future extension.

Figure 3 shows a view of the video system architecture designed with the SoftArch tool (1); this view's corresponding Pounamu view XML format (2); and the result GXL file using the developed converter (3). This architecture visualisation is a simple static structure view and one that we may wish to view in a variety of different software and interface tools e.g. SoftArch, Pounamu, Argo/UML, an SVG or VRML web browser plug-in. The GXL and Pounamu XML formats are quite similar in structure and purpose so implementing these converters was relatively straightforward. However, when importing GXL-described views into Pounamu views a set of graph layout defaults must be added to the newly generated Pounamu view model as these do not exist in many GXL descriptions.

This layout generation was implemented using a Java algorithm using the DOM interface provided by Java's XML parser to enable standardized access to the structure of the imported Pounamu XML-format view documents. Java was used so that complex computation could be used and because some information in Pounamu has no direct relation to GXL but may used by some tools (e.g. shape design, position or size). To support these tools we divided the document into different parts (GXL and properties) and added references in the GXL document to the corresponding property file. When exporting Pounamu XML format views to GXL we leave the Pounamu layout information in the GXL format. As this isn't used by some 3rd party GXL-based tools it may be ignored and lost if the data is re-imported into Pounamu or other tools.

Our GXL import/export converters for Pounamu enable any Pounamu-defined visual notation views to be converted to and from GXL representations. Thus any GXL-compliant software or graph-manipulation tool can consume and/or produce information that can be viewed and edited within our Pounamu meta-tool environment. New visualisation notations designed in Pounamu and view instances of these notations can be exchanged with these tools via the GXL-based common exchange format.

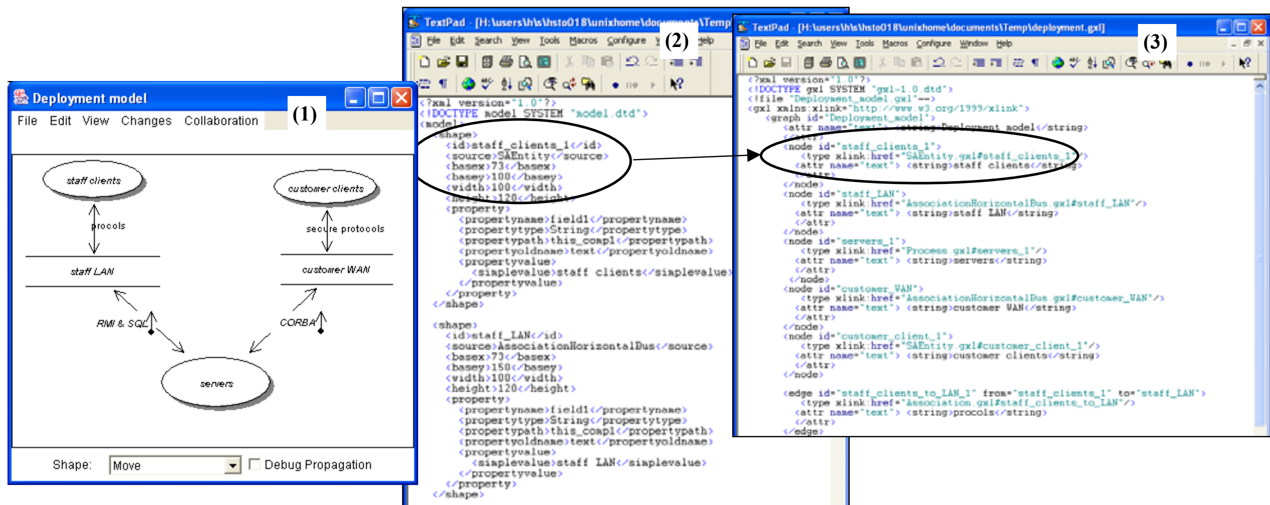

Figure 3: An example of converting a Pounamu XML format architecture visualization into GXL.

4.2 GXL to SVG Converter

SVG (Scalable Vector Graphics) is a recommendation of the W3C group [22]. The main goal of the SVG format is to describe two-dimensional graphics in XML. Our interest in SVG was as a vehicle to provide developers a thin client interface with limited interaction. Currently Pounamu provides a thick-client interface to editing views of software information e.g. the software architecture diagrams shown previously are all viewed and edited via a desktop environment. The availability of SVG as a plug-in for every common browser makes it a very portable front-end for viewing and/or editing graph-based information visualisation notations, including those for software tools. A developer could for example use an SVG plug-in in a browser to view models designed with Pounamu without having the Pounamu system being locally installed.

A very limited converter from GXL to SVG already exists and we initially attempted to adapt this to allow Pounamu views, using GXL as intermediate format to be converted to SVG for viewing in browsers [11]. However a major drawback of this converter proved to be a lack of support for hierarchies (graphs inside graphs) and this is an important issue for a breakdown of a complex systems. Our example software architecture visualisation notations all use limited forms of this [7]. In addition, the implemented layout algorithm in this 3rd party GXL to SVG converter is only able to arrange all nodes in a circle, which is extremely unsatisfactory for many applications such as our software architecture visualisation notations. Converting model information to SVG with this converter also loses some information e.g. directions of the edges and links between nodes.

To overcome these problems we have developed a new GXL to SVG converter that enables hierarchical graph-based notations to be converted, uses a more flexible layout algorithm, and preserves more of the GXL-described notation characteristics in the SVG format. Figure 4 shows the result of converting a view of the video system architecture to SVG and viewing this SVG-format software architecture diagram in a web browser with an SVG plug-in. This was a Pounamu view initially converted into GXL by our Pounamu GXL converter.

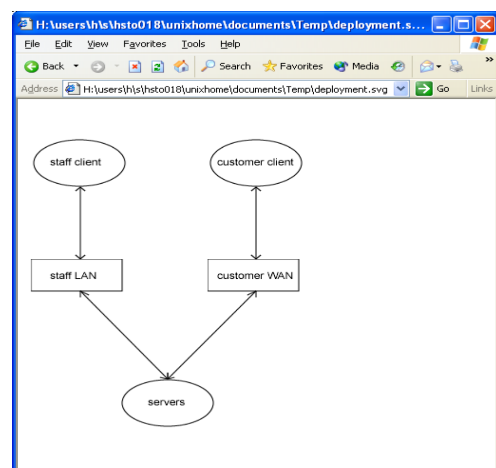

Figure 4: Example of an SVG software architecture view in web browser converted from a GXL format.

A number of further complexities arose when implementing this notation converter. Since SVG is a vector format, calculations have to be done to arrange the shapes appropriately, a quite different approach to the GXL and Pounamu (and most other graph description formats). We also had to convert the GXL data format

into SVG data format, rearranging the structure of the XML data significantly in places. The generated SVG-format software visualisations can then be viewed in a web browser using a suitable SVG plug-in. Our converter from GXL to SVG can be accessed via a URL and runs as a servlet to enable distributed users to access the architecture diagrams from their browsers.

We have only implemented a converter from GXL to SVG, since the plug-ins are view only and Pounamu users interact with the SVG-format visualisations via browser-based scripting. We have not investigated converting SVG into GXL or Pounamu's own view XML format.

4.3 GXL to VRML Converter

X3D is a XML-based standard notation for defining interactive web- and broadcast-based 3D content integrated with multimedia [19]. X3D is the successor of VRML [20], the original ISO standard for web-based 3D graphics and extends it with new features, additional data encoding formats, stricter conformance, and a component based architecture allowing a more modular approach. X3D is intended for use on a variety of hardware devices and in a broad range of application areas such as engineering, multimedia presentations, and shared virtual worlds.

Our aim in building a converter from the GXL format to the X3D format was to provide developers a 3D-based static and dynamic view of software system information. This was in contrast to hand-coding into Pounamu its own 3D views e.g. using Java3D graphics libraries, which would be an enormous effort. In our software architecture examples, we may wish to view large, complex static architecture dependencies and other structural relationships using VRML-style 3D virtual environments. We may also want to view complex dynamic information about a software architecture, such as architecture performance measurements, using a 3D approach (e.g. the number of calls can be expressed in the size of shapes; requests across a network link can be expressed by thickness or colour of links between client and server nodes and so on). A key advantage of such visualisations using 3D virtual reality environments is that navigation of the complex models is more intuitive, via direct manipulation of the environment, zoom in and zoom out and three-axis rotations.

Figure 5 shows a visualization of a software architecture in a VRML web browser plug-in. This uses colour to show the frequency of function calls related to various architecture components of the video system. The designer can rotate the models in 3 dimensions; can zoom in and out click on nodes/links to request detailed information about the performance measures and the architecture components to be shown. In previous work, we tried to show such information using 2 dimensional views in SoftArch, and user feedback indicated difficulty in navigating and interpreting the information [6].

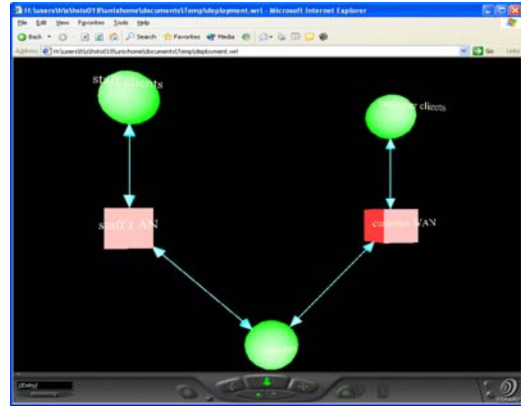

Figure 5: Example of a software architecture diagram converted to VRML from GXL.

We implemented the GXL to VRML conversion by building a Java-implemented converter that translates GXL nodes and links into VRML 3D scene description elements. This converter also performs some simple layout to ensure that the resulting scene elements are separated using some basic heuristics, but uses the Pounamu-added layout information added to the GXL by our Pounamu-to-GXL converter. For simple architecture visualisations based on the GXL representation the same layout algorithms as for two dimensions can be utilised. For very complex visualisations, use of the extra dimension in VRML needs to be made to improve the complexity of the view. We have only made very basic use of this third dimension in our converter to date. Another feature of VRML and X3D we could make further use of is control of viewing and navigation via proximity sensors and scene interaction event-handling. The scripting extension of X3D allows a wide range of interactions to be defined for the visualisation, and can be used in a similar manner to SVG scripting to allow interaction with Pounamu from the X3D views. This interaction can include manipulating the underlying visualisation data and requesting other views to be shown.

5. Notation Converter Generation

Motivated by the similarities in many converters and the effort involved in developing them, we are currently developing a general approach for modelling the conversion between different notations. Our approach is being realised as a Java-based converter-generator framework that will enable developers to describe inter-notation mappings and have suitable visualisation notation converters generated for them. The architecture consists of two tools. A Unified Notation Mapping (UNM) specification language provides a notation format

specification facility along with a notation mapping specification syntax, allowing developers to describe a mapping from one notation to another. An Automatic LAnguage MApper TOol (ALAMATO) is used to translate UNM inter-notation mapping specifications into specific notation-to-notation converter implementations.

5.1 A Unified Notation Mapping Language

Figure 6 shows the basic structure of UNM. It supports the description of a visualisation notation (1), in terms of fundamental notation elements and their inter-relationships. UNM supports text-based file formats (we assume each visual notation has a textual "save" format) and generates, with the interactive help of the developer, a dictionary (2) which specifies the syntactic structure of the notation. This dictionary is used by ALAMATO to generate a converter between two different visualisation notations. The dictionary structure can be annotated by editors (3) to specify additional behaviours, with the resulting structures represented as Java code (4). UNM is also used to create a mapping table (5), to provide ALAMATO with the required mapping information for each notation.

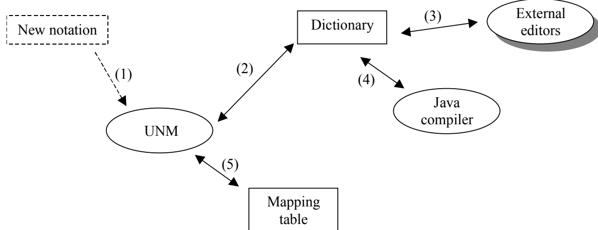

Figure 6: UNM and its components.

Storing the mapping information separate from the dictionary has the advantage of more freedom of mapping. Different users may want different mappings for specific languages, e.g. one user wants to have a GXL node to SVG mapped as a rectangle and another user prefers another shape. The user has only to change the mapping table, not the more complex dictionary which contains the implementation part of the notation and how each element in the notation is textually or non-textually represented. External editors extend the existing functionality, for example doing boundary checking or setting special dynamic behaviours (in the simplest form adding date or time to an output notation), and can be used for more complex functions such as layout generation. External editors can also be used for including notations not viewable directly in UNM, such as graphic formats (GIF, JPEG) or text document formats (PDF, MS Word documents). For this, UNM generates a skeleton and the functionality implemented using an editor tool.

Figure 7 shows a sample dictionary containing several notations. The user designs this hierarchy in UNM and UNM automatically assigns the corresponding tags used for this node. These tags can be used to both find the node in input files and also later to create the output file.

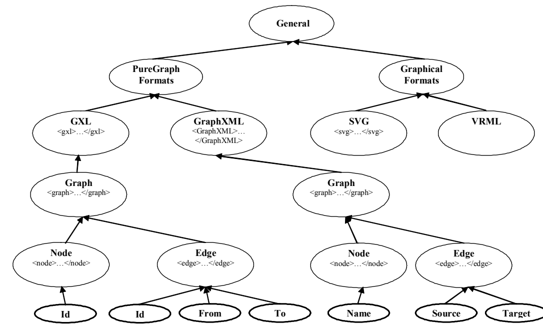

Figure 7: Part of a UNM dictionary.

The dictionary consists of two types of nodes, structure nodes and content nodes. Structure nodes provide dictionary structure and categorize branches. For example, in Figure 7 the nodes *PureGraphFormats* and *GraphicalFormats*. Content nodes have a value attribute, which contains the corresponding value from a tag. External editors can be used to annotate each node. A node can also contain a list of nodes, for example to support structures similar to a number of rectangles embedded in a canvas. Node *Id* in Figure 7 with bold border indicates a required flag. If this information cannot obtained (either by mapping or calculated) the parent node will not be mapped into the output format. This is sometimes reasonable e.g. when a node without identification cannot be referred to elsewhere.

An example of using UNM is shown in Figure 8. This is a small part of a mapping table for GraphXML, GXL, SVG and VRML, being specified in our prototype UMN tool. Mappings can be designed using UNM or by editing the generated mapping table code. This table gives ALAMATO information about how the nodes are to be mapped between notations. Virtual nodes can be used, which provide no output but categorize the mappings, for example the node *Shape* (1). Another type of node enables links between branches and guarantees only a single child node in another branch. For example, it is not possible to assign another child node to the link node *NodeShape* based in the *Shape* branch (2). With this construct it is explicitly defined that *Node*s from GXL or GraphXML have to be mapped to SVG or VRML as rectangles. Each node in this mapping table is derived either from *MultipleValuesNode* or *PrimitiveValueNode* (3). Both are derived from a *GeneralNode* which provides basic functionality required by every node like analysing the textual form of the nodes, set and get values, set a required flag and so on. *PrimitiveValueNode* extends the general functionality by adding behaviour required to create the textual structure of primitive nodes. In contrast to *PrimitiveValueNode* the *MultipleValuesNode* includes more complex operations on nested nodes.

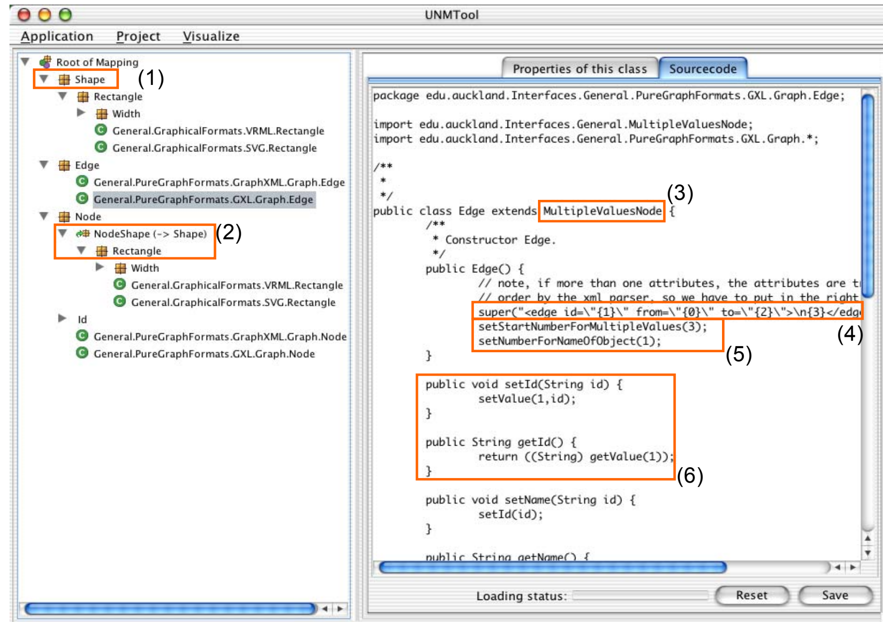

Figure 8: Example of a mapping table in our UNM specification tool.

In the constructor of each node the textual representation is specified (4) which is used by *GeneralNode* to construct its textual form and identify the parameters of a node. In Figure 8 it is also shown how to specify in the framework more information about each node (5). For example with *SetStartNumberForMultiple Values* the position is explicitly defined, where (in the textual representation) additional nodes have to be inserted. Another example of functionality provided by the UNM framework is to specify the unique name of nodes, which is specified by *setNumberForName OfObject*, where the number again is related to the position in the textual form. Using this, UNM has similar navigation facilities as DOM to retrieve documents for specific type of nodes or (object) nodes. Additionally the developer can extend the node implementation as in Figure 8 - *setId* and *getId* are used to improve the readability (6).

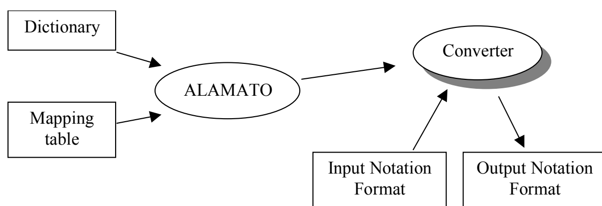

Figure 9: ALAMTO converter generator approach.

5.2 ALAMATO

The architecture of our ALAMATO (Automatic LAnguage MApper TOol) is shown in Figure 9. This tool generates a custom inter-notation mapping converter to translate one visualisation notation format into another. It uses the class hierarchy and mapping table generated by UNM to produce a parser that reads a source visualisation notation format and builds up a tree structure for the input notation. It then uses the UNM-specified mapping scheme to generate the structure and element conversion from the input notation to the output notation. The output visualisation notation information is then formatted into the output format for consumption by another tool.

6. Discussion

Over many years of research into software visualization and the development of a wide variety of software visualization-supporting tools we have identified the need to support inter-notation translation i.e. the exchange of view (or visualization)-level information between tools, not just model-based information exchange. This is harder in some respects to pure model-based tool integration [9, 17] which focuses on translating between descriptions of software information formats. Translating between notations requires the mapping of descriptions of complex views, which make use of a very wide variety of boxes, lines, colour, positioning, text characteristics, annotations and possibly 3D structures. A variety of graph-based converters have been developed [3, 11, 13, 21]. Unfortunately we found that many of these conversion tools to only partially implement visual notation mappings. Many of these converters are “lossy” and remove source notation information that can not be represented in the target notation format e.g. layout when translating into GXL. We also found that modifying them was usually very difficult due to the low-level

programming required. In addition, in some instances a tool developer will need to provide default information between notations e.g. layout, icon appearance that isn't in the source notation information. Often existing converters don't support this or use inappropriate defaults.

Some of the complexities that need to be handled in inter-tool visual notation conversions include supporting translation between quite different notation description structures (e.g. between GXL and SVG), the defaulting of values particularly relating to layout (2D or 3D) of visualizations (e.g. Pounamu to VRML), and the translation of simple editing/hyper-linking support into scripting languages or other semantic editing/viewing control (e.g. GXL to SVG and VRML). Our Universal Notation Mapping language aims to capture these notation mapping complexities so that software tool developers can specify inter-notation mappings at high levels of abstraction. Our ALAMATO converter generator framework processes these mapping specifications to automate the conversion of notation encodings. Where appropriate we aim to add generation of editing action conversions (i.e. view editing and navigation interactions) between different software tools. This would allow developers to readily exchange software visualization notations (appearance – view syntax) as well as interaction behaviour (editing – view semantics) between software tools and 3rd party visualization products.

7. Summary

We have developed a number of software visualization converters that support the translation of notation instance descriptions between a range of formats e.g. our custom Pounamu XML format and the general GXL format; GXL and SVG; GXL and VRML; and have investigated translation between GXL and Excel chart, Visio diagram and GXL and GIF/image map renderings. From our experiences building these visual notation converters we have designed a converter generator framework that allows tool developers to specify inter-notation mappings and to have custom visualization notation converters generated from these mapping specifications. We have developed a Universal Notation Mapping language and prototype notation specification tool and are developing an automatic converter generator from these specifications.

References

1. Beaumont, M. and Jackson, D. Visualising Complex Control Flow. In *1998 IEEE Symposium on Visual Languages*, Halifax, Canada, September 1998, IEEE.
2. Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modelling Language User Guide*, Addison-Wesley, 1999.
3. GCF – a GXL Converter Framework. http://www2. informatik.unibw-muenchen.de/GXL/triebsees/.
4. Egyed, A. and Kruchten, P., Rose/Architect: a tool to visualize architecture, In *Proceedings of the 32nd Hawaii International Conference on System Sciences,* January 1999, IEEE CS Press.
5. Grundy, J.C. Software Architecture Modelling, Analysis and Implementation with SoftArch, In *Proceedings of the 34th Hawaii International Conference on System Sciences (Software Architecture Mini-track)*, Maui, Hawaii, IEEE CS Press.
6. Grundy, J.C., Cai, Y. and Liu, A. Generation of Distributed System Test-beds from High-level Software Architecture Description, In *Proceedings of the 16th International Conference on Automated Software Engineering*, San Diego, IEEE CS Press, pp. 192-200.
7. Grundy, J.C. and Hosking, J.G. High-level Static and Dynamic Visualisation of Software Architectures, In *Proceedings of the 2000 IEEE Symposium on Visual Languages*, Seattle, Washington, Sept. 2000.
8. Grundy, J.C., Mugridge, W.B., Hosking J.G. and Kendal, P. Generating EDI Message Translations from Visual Specifications, In *Proceedings of the 16th International Conference on Automated Software Engineering*, San Diego, IEEE CS Press, pp 35-42.
9. Grundy, J.C. and Hosking, J.G. Software Tools, *Wiley Encycolpedia of Software Engineering*, 2nd Edition, Wiley, December 2001.
10. GXL (1.0) Tools. http://www.gupro.de/GXL/tools /tools.html.
11. GXL2SVG. Example of the JGraph project. http://jgraph.sourceforge.net/downloads.html.
12. Hill, T., Noble, J. Visualizing Implicit Structure in Java Object Graphs, In *Proceedings of SoftVis'99*, Sydney, Australia, Dec 5-6 1999.
13. Hold, R.C., Winter, A., and Schürr, A. GXL: Toward a Standard Exchange Format, In *7th Working Conference on Reverse Engineering*, IEEE CS Press, 2000.
14. Liu, A. Dynamic Distributed Software Architecture Design with PARSE-DAT, In *Proceedings of Software – Methods and Tools,* Wollongong, Australia, November 2000, IEEE CS Press.
15. Quantrani, T. *Visual Modeling With Rational Rose and UML*, Addison-Wesley, 1998.
16. Reiss, S.P. A framework for abstract 3-D visualization, In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE CS Press.
17. Robbins, J. and Hilbert, D.F. Extending design environments to software architecture design, Automated Software Engineering, Vol. 5, No. 3, July 1998, pp. 261-390.
18. Stankovic, N. and Zhang, K. Towards Visual Development of Message-Passing Programs, In *Proceedings of 1997 IEEE Symposium on Visual Languages*, IEEE CS Press.
19. The Web3D Consortium. Extensible 3d (x3d) graphics. http://www.web3d.org/x3d.html.
20. The Web3D Consortium. The virtual reality modeling language. http://www.web3d.org/Specifications/VRML97/.
21. Winter, A. Exchanging Graphs with GXL, Graph Drawing – 9th International Symposium, GD 2001, Vienna
22. World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.0 Specification. http://www.w3.org/ TR/SVG/.