Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 211 (2008) 171–179
www.elsevier.com/locate/entcs

On Challenges for a Graphical Transformation Notation and the UMLX Approach

Edward D. Willink[1]

GMT Consortium
www.eclipse.org[2]

Abstract

Freely available experimental transformation languages have begun to stimulate practical usage of textual transformation notations. The forthcoming QVT transformation languages may provide standardisation or at least interchange capabilities for these experimental languages. Graphical transformation notations are proving rather less successful. We identify many disadvantages of the graphical approach, consider how they can be circumvented and describe changes in the UMLX notation and tool support to improve usability and QVT compatibility.

Keywords: Model Transformation, Transformation Editor, Graphical Transformation Notation.

1 Introduction

Model transformations are becoming steadily more practical and rigorous with the advent of better meta-model based tools such as ATL [7] or Tefkat [6]. The transformation language for each of these tools is proprietary and so inhibits the wider exploitation of these transformations. The increasingly imminent QVT standard [8] for a suite of three languages may avoid incompatibility problems, possibly by rendering the existing languages obsolete, more likely by defining an interchange point so that transformations for language A may be transformed to a QVT language and from there to language B.

The current transformation languages and the QVT submission are largely textual in syntax, which is perhaps surprising given that they operate on meta-models that are often drawn using a graphical style derived from UML.

A graphical notation can be visually attractive and this provides a significant advantage over a textual notation. An interesting picture may provoke fruitful dis-

[1] Email: EdWillink@iee.org

[2] The author is not associated with the QVT-merge consortium of which his employer, Thales Research and Technology (UK) Ltd, Reading, England is indirectly a part.

doi:10.1016/j.entcs.2008.04.039

cussion within a team or with passers by. Program text often has a more restricted readership.

In this paper we try to understand why graphical transformation notations are proving less successful and consider when and how this lack of success should be remedied. We first consider the generic advantages and disadvantages of textual and graphical notations, before examining the reasons why some graphical notations have proved successful and others have not. In Section 2 we examine a variety of graphical transformation notations in greater detail, noting their similarities with respect to the use of declarative patterns in principle, but significant differences in practice and even greater differences with respect to transformation rules. Then in Section 3 we outline the tool support for UMLX and notational enhancements to improve alignment with QVT.

1.1 Textual and Graphical Notations and Tools

A textual notation for a language has many advantages over a graphical notation. Tooling requirements are limited and so a wide variety of editors can be used. Printing or viewing is straightforward and activities such as searching or comparing pose few challenges. Even when more sophisticated syntax-sensitive editing is desired, there are a number of customisable editors to choose from.

A textual notation does not suffer particularly from scalability issues. The use of multiple source files and hierarchical language constructs enables very large overall line counts to be managed. Unduly long lines are readily avoided at the expense of a slightly increased line count.

In contrast, a graphical notation has many disadvantages. Specialised editors are required often using file formats with limited scope for interchange with alternative editors. Ancillary tasks such as printing require similarly specialised tooling that consequently must form part of the editor. The extra pagination and rendering complexities are not always satisfactorily realised for an adequate range of print media. Other activities, such as searching also rely on the specialised editor. With so much reliance on a specialised editor, the choice of editors is limited, and very limited if significant financial investment is to be avoided.

Graphical notations may also suffer from scalability; there is a limit to the amount of zooming that can be tolerated to enable a diagram to fit on a single sheet of paper. This is a major problem when the notation fails to represent a reasonably sized and sensibly modularised problem on a single sheet, and a minor problem if navigation between diagrams is cumbersome.

1.2 Successful Graphical Notations

A graphical notation must offer significant benefits to overcome these major disadvantages, so it is worth reviewing where and why graphical notations are successful.

A textual language often aligns the vertical dimension with a sequential order, so that earlier lines precede later lines in execution order, or spatial layout. This suits the procedural style of programming that is so widespread through the dominance

of languages such as C or Java. A critical weakness in textual languages arises when a more declarative perspective is taken. It is difficult to identify all uses of a particular concept such as a variable, so it is necessary to search the code for all occurrences of a name.

A graphical notation imposes no inherent layout constraints. This freedom allows some very poor diagrams to be drawn when little attention is paid to ease of understanding. This does not matter for diagrams drawn on the 'back of an envelope', or a whiteboard, where the audience is live and the sole purpose of the diagram is to clarify understanding or intention. These diagrams should then be destroyed. Diagrams that are to be preserved should be drawn with much greater discipline, so that a cold audience can easily grasp their meaning. It is often helpful to observe some form of left to right and/or top to bottom discipline to ease comprehension by providing the audience with a starting point. A few well drawn diagrams may provoke useful discussions about a design approach, whereas textual representations are less amenable to casual review.

The lack of an inherent layout makes graphics well suited to a declarative exposition of many interrelated concepts. Each concept is a graph node denoted by a symbol, and each interrelationship is a graph edge denoted by a line. The lines between symbols are easy to identify and so all relationships involving a particular node are easily determined.

With this insight, it is not surprising that State Machines have been so successful graphically, since the many peer states can be shown with equal or weighted import. The presence and absence of transitions between states is very evident. For similar reasons, Entity Relationship Diagrams or Class Diagrams are also useful to enable the structural relationship between diverse data elements to be visualised.

Graphical notations for functional and dynamic relationships are often less successful. Program Flow Charts were popular in 1970s but are now little used. The Schlaer-Mellor [9] notation for a Data Flow Diagram involves lines for flows between processes and stores; this notation has insufficient precision to be used as more than an overview. On the other hand, the SDL [4] notation has a rich set of symbols for expressing sequential computations. This notation can have excessive precision; the graphics appears more complicated than comparable text.

A variety of forms of Block Diagram have been used for decades in the Electronics industry. Many of these diagrams have insufficient precision for code generation but could be given that precision once adequate, quite possibly MDA-related, tools mature. These diagrams are a declarative exposition of a variety of potentially concurrent activities with some similarities to the revised Activity Diagram in UML2.

The above selection of usages from a variety of fields demonstrates that graphical notations succeed for declarative problems but fail for procedural problems. Transformations based on UML-like meta-models might be expected to continue the graphical tradition of class diagrams and favour a graphical notation. However the extended functionality for imperative transformation languages is procedural, an area where UML proves less satisfactory.

1.3 Utility

In addition to the handicaps that may arise from the difficulties of providing a suitable graphical presentation, further difficulties may arise from a poor notation.

If a notation lacks obvious meaning, the utility of the notation as a stimulus for informal review and discussion may be impaired, and potential users may be discouraged by a need for extensive training. For transformation authors, the learning cost may be ameliorated by good tool support, but for casual reviewers the notation must be as obvious as possible.

A new notation should offer obvious and genuine advantages over alternatives. Graphical notations can have a clear advantage in appearing at least superficially attractive and may back this up by providing compact notation and easy to use tooling. But genuine advantages require a notation to have as much rigour as possible, so that as many different forms of error are eliminated by the design, thereby improving the productivity of the notation. Where possible, this rigour should be hidden from practical programmers who may be discouraged by the too overt appearance of, for example, set theory. Although, with OCL 2 becoming an essential part of so many transformation approaches, it may be that programmers will be forced to extend their mathematical background.

1.4 Conclusions

¿From the above we may conclude that a graphical notation can be superior when it provides a declarative exposition. To satisfy the potential for providing diagrams for casual review, the notation should be clear, precise, compact and easy to understand, yet rich enough to express realistic problems.

A graphical notation should provide sufficient precision to ensure that diagrams are useful, but should recognise that not everything is sensibly presented graphically. The notation must therefore co-exist with a textual form so that users have a free choice to use whichever of textual or graphical expositions is clearest.

The scalability issues for a graphical notation should be addressed by ensuring that a diagram can be modularised sufficiently to fit on a page and in the tool support by providing good navigation between pages.

The underlying principles behind the notation should be rigorous to assist in provision of portable, reliable, maintainable and re-usable transformations.

The notation should be easy to learn, fun to use and well supported by tools to provide a pleasant programming environment. Support by free tools may serve to increase the user base more rapidly.

2 Graphical Transformation Notation

2.1 Relationships and Instances

It is reasonable to assume that anyone using meta-model transformations is familiar with UML, and so basic UML concepts such as composition require less learning

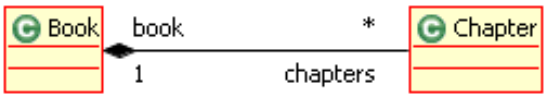

Fig. 1. UMLX Composition Definition.

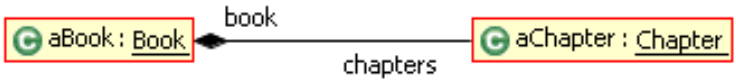

Fig. 2. UMLX Composition Instance.

than their textual counterpart.

Figure 1 [3] shows a simple composition in which a `Book` may contain zero or more `Chapters`. A `Chapter` must be contained by exactly one `Book`, that may be accessed as the `book` property of the chapter.

UML practitioners might also recognise Figure 2 that shows a similar relationship between two objects rather than two classes. A particular instance of a `Book` is identified as `aBook` and contains an instance of a `Chapter` identified as `aChapter`.

In traditional UML usage, `aBook` and `aChapter` are instances whose identification facilitates exposition of the remainder of the UML design; there is a single {`aBook`, `aChapter`} tuple for the whole design.

In model transformation usage, the object diagram is re-interpreted to define a pattern, so that wherever the pattern is satisfied, an appropriate transformation rule can be activated. `aBook` and `aChapter` are therefore variables within the pattern that bind to instances in the model to be transformed; there is a distinct {`aBook`, `aChapter`} tuple for each possible rule activation for the transformation.

The relationship between `aBook` and `aChapter` is not shown in a textual transformation language; it is implicit in the declaration of two variables with types from the meta-model comprising `Book` and `Chapter`. This implicit relationship avoids typing, but inhibits casual review by readers who are unfamiliar with the meta-model and may lead to obscure error messages for programmers who misunderstand it.

2.2 Presentation

Although graphical transformation notations have adopted the style of Figure 2 there has been some divergence in their presentation and elaboration.

In the revised merged QVT submission [8], the class name underlines and the line decorations are omitted. Omission of the underline is a minor stylistic deviation from UML. Omission of the line decorations deprives the reader of the distinction between composition and association and the disambiguation of multiple associations involving the same classes. AGG [3] uses a more conventional Graph Transformation notation and so underlines and line decorations are again omitted, and instance names are replaced by instance numbers.

[3] The diagrams in this paper are drawn using UMLX, which is based on and follows the Eclipse/GEF EDiagram example in adding an icon at the top left of symbols.

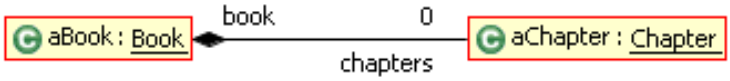

Fig. 3. UMLX Composition Instance With Zero Multiplicity.

Gmorph [10] and GRE [2] are much closer to UMLX [12]. Gmorph underlines both instance and class name while GRE uses a stereotype notation [4] for the class name.

2.3 Multiplicity

UML provides three relevant decorations for relationships; the multiplicity, the property name, and the diamond for composition. The last two of these can assist in understanding the correspondence between pattern and meta-model. Re-use of multiplicity in this purpose would be confusing, and so some graphical notations such as the QVT submission just omit the multiplicity. Figure 2 therefore denotes the relationship between a single `aBook` and a single `aChapter`.

GRE showed multiplicity in the style of UML but did not exploit it. Its successor, GReAT, recognised that non-unit multiplicity could support patterns involving sets of objects rather than just objects. The corollaries of this interpretation are discussed in [1]; a pattern could now specify that its rule was applicable only to each book containing two or more chapters, rather than just to each book (that might contain some chapters). However, in GReAT, the multiplicity was shown as an instance stereotype thereby identifying the absolute size of the set of matched objects for each free variable, rather than the relative size of the set of matched objects between the ends of the decorated relationship.

UMLX followed GRE in using the UML presentational style for multiplicity and used the UML multiobject notation where a set of objects rather than just an object was bound to a variable. UMLX followed the disciplined principles of GReAT to define the meanings of these sets. UMLX further recognised that a zero multiplicity signified negation, so that a simple zero as in Figure 3 denotes a pattern that matches each `Book` that contains no `Chapter`s. Other notations have introduced a distinct crossing-out symbol for this purpose.

The QVT submission also exploits the UML multiobject to display sets of objects, however the submission has limited capabilities for patterns that involve sets of objects. UMLX has a more comprehensive capability defined in [11]. The GReAT analysis of multiplicity identifies some limitations in its use. For example, in a complex pattern in which there is a cycle, interpretation of the cycle clockwise may lead to a different meaning to the anticlockwise interpretation. This is clearly unacceptable and must be reported as an error. The problem is analogous to the need for parentheses to impose an intended meaning on an arbitrary expression involving 'and' and 'not' operators. The graphical equivalent of a parenthesis requires an explicit grouping of part of the cycle so that there is a single relationship between

[4] Text between angle brackets.

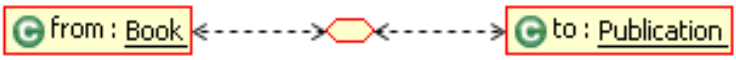

Fig. 4. UMLX Evolution.

the residual part of the cycle and the grouped part. Provision and evaluation of this grouping as a graphical facility is work in progress for UMLX.

2.4 Rules

The principle of using patterns to define the application context of a transformation rule is common to perhaps all graphical transformation approaches. The divergence between approaches arises in the relation between input and output context.

GRE showed separate input and output patterns with creation relationships between them.

UMLX was inspired by GRE but replaced the creation relationship by declarative preservation and evolution relationships. A preservation relationship keeps the input element for re-use on as an output element. An evolution relationship may add an output element or elements with respect to an input element or elements, and may also delete an input element or elements with respect to an output element or elements. Preservation extends the Keep operation of Graph Theory [5], to support keeping not just a node, but also all its composed descendants. Evolution combines Add and Delete operations in a multi-directional relationship that always defines a traceability relationship.

GReAT abandoned the potentially declarative characteristics of GRE to pursue an imperative approach. This achieves layout economy by overlaying shared input and objects but the use of colours to distinguish input-only, output-only and shared objects lacks intuition and cannot be rendered in black and white. Imperative operators are graphically sequenced to define the transformation. The result is an unfortunate mix of imperative and declarative meaning in the same diagram.

The proposed graphical syntax for QVT makes no attempt to relate input and output patterns visually; each is drawn independently, with the relationship between them defined in a textual region by OCL expressions involving the names of input and output objects.

Gmorph also relies on text to associate independent input and output graphics. AGG uses shared instance numbers to associate independent input and output graphs.

3 UMLX Tooling

We have already described how UMLX attempts to align with the goal of a providing a declarative and easy to understand notation by re-using the familiar UML notation for defining patterns, extended semantically by a solid definition of the meaning of multiplicity and sets of objects. A declarative graphical extension supports definition of the transformation between the patterns with the need to resort

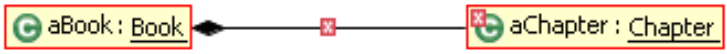

Fig. 5. UMLX Composition Instance With Errors.

to text.

In order to fulfil the ease of use criterion, the original implementation of UMLX based on GME and inflexible Microsoft technologies, has been replaced by Eclipse plug-ins based upon GEF and more particularly its EDiagram example that uses EMF for meta-models. This provides a free, portable and standard integration platform. Sadly, it exchanges GME's disciplined meta-modelled approach to defining a graphics editor by the much more flexible but rather manual code cutting capabilities of GEF, EMF and Java. It is hoped that GMF may provide the best of both worlds for the next revision.

The Eclipse support for UMLX exploits an Outline view of one or more meta-models to provide a dynamic palette of Drag and Drop elements that can be dropped onto a sheet to instantiate, or onto a graphical element to re-instantiate, a legal design element. Widespread use of the Outline, Drag and Drop, in-place editing and standard GEF editing interactions enables designs to be built up with considerable ease.

An editor supporting graphical transformations should assist the user in drawing transformations that comply with their meta-model, so the decorations are supplied automatically whenever an unambiguous relationship is drawn between two classes. When an illegal relationship is drawn, the problem is shown as in Figure 5 which shows the impact on Figure 2 following deletion of `Chapter` and the `Book` to `Chapter` relationship from the meta-model. The pattern is still syntactically correct but because the meta-model has been changed, the pattern is no longer semantically valid. A similar problem arises in textual languages when a subroutine call is invalidated by a change to its signature. Each inappropriate graphical element shows a red Eclipse error marker. These also appear in the Eclipse Problem, Outline and Resource views, so that the graphical markers behave in a similar way to textual error markers in the Java editor.

The editor supports partitioning a design into sheets using three different diagram types. Meta-Model Diagrams support maintenance of Ecore meta-models, that are instantiated within Transformation Rule Diagrams where the UMLX transformations are drawn. A further Transformation Context Diagram supports aggregating many UMLX-defined or QVT-defined rules as part of a QVT compatible Transformation.

4 Summary

The relative characteristics of textual and graphical notations have been contrasted first from a relatively generic point of view and then by contrasting a number of different graphical transformation notations.

The many advantages of a textual notation suggest that a graphical notation is

most likely to be superior when it uses a declarative style that is easily understood by casual reviewers. With a graphical notation that is superior in some contexts, a transformation programmer may then be offered a choice of notations so that the most appropriate can be chosen for each context.

UMLX complies with these declarative perspectives. It is hoped that the revised Eclipse-based tooling available from http://www.eclipse.org/gmt will provide a friendly easy to use and productive environment that will encourage the use of declarative transformations.

References

[1] Agrawal, A., Levendovszky, T., Sprinkler, J., Shi, F., Karsai, G.: Generative Programming via Graph Transformations in the Model-Driven Architecture, OOPSLA 2002 Workshop on Generative Techniques in the context of Model Driven Architecture, November 2002 URL: http://www.softmetaware.com/oopsla2002/karsaig.pdf

[2] Agrawal, A., Karsai, G., Shi, F.: A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations, URL: http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_0_0_2003_A_UML_base.pdf

[3] Buttner, F., and Gogolla, M.: Realizing UML Metamodel Transformations with AGG In: Proceedings of the 4th International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT 2004, Barcelona, Spain, March 2004, URL: http://wwwcs.uni-paderborn.de/cs/ag-engels/GT-VMT04/GT-VMT04-camera-ready.zip

[4] CCITT Recommendation Z.100: Specification and Description Language SDL, Annexes A-F to Z.100. 1988. Blue Book, Volume VI.20 - VI.24, ITU, General Secretariat- Sales Section, Places des Nations, CH-1211 Geneva 20.

[5] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach, In Rozenberg, G., ed., The Handbook of Graph Grammars, Volume 1, Foundations, World Scientific, 1996.

[6] DSTC QVT Team: Tefkat, URL: http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/

[7] Jouault, F., and Kurtev, I.: Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, October 2005, URL: http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev__transforming_models_with_atl.pdf

[8] Revised submission for MOF 2.0 Query/View/Transformation RFP (ad/2002-04-10), OMG Document, ad/2005-03-02, URL: http://www.omg.org/docs/ad/05-03-02.pdf

[9] Schlaer, S., and Mellor, S.: Object-Oriented Systems Analysis: Modeling the World in Data, Yourdon Press, 1988.

[10] Sendall, S.: Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?, OOPSLA 03 Workshop Generative techniques in the context of MDA, 2003 URL: http://cui.unige.ch/~sendall/files/sendall-mda-workshop-OOPSLA03.pdf

[11] Willink, E.: Towards a Formalization of UMLX, URL: http://dev.eclipse.org/viewcvs/indextech.cgi/*checkout*/gmt-home/subprojects/UMLX/doc/UmlxFormalization/UmlxFormalization.pdf

[12] Willink, E.: UMLX : A Graphical Transformation Language for MDA, URL: http://dev.eclipse.org/viewcvs/indextech.cgi/*checkout*/gmt-home/doc/umlx/Oopsla2003.pdf