Electronic Communications of the EASST
Volume 4 (2006)

Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT 2006)

Model Instantiation and Type Checking in UMLX

Edward D.Willink

13 Pages

Guest Editors: Gabor Karsai, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Model Instantiation and Type Checking in UMLX

Edward D.Willink

EdWillink@iee.org
Eclipse GMT Project

Abstract: OMG's MDA initiative encourages the use of meta-model based transformations and re-usable specifications. We discuss how Graphical Transformation Notations such as UMLX reduce opportunities for errors in this programming domain.

Keywords: MDA, Model Transformation, Graphical Transformation Notation

1 Introduction

Software technology has had made many advances since Fred Brooks' *There is no silver bullet* [4]. Each advance has made certain types of errors hard, if not impossible, to make but sadly fails to confound Fred Brooks' pessimistic viewpoint.

The Model Driven Architecture with its emphasis on transformation of models from comparatively abstract Platform Independent Models to comparatively concrete Platform Specific Models provides a further significant advance.

When the MDA approach is adopted enthusiastically to support progressive transformation from very abstract Domain Specific (Visual) Language models to very concrete models in the form of executable code, optimists may hope to finally make some inroads on software portability, reliability and cost. Once models are sufficiently abstract to escape from portability issues, sufficiently large to represent subsystems worthy of re-use, and sufficiently parameterisable to adapt to a desired set of performance characteristics, we may be able to define re-usable systems. Once transformations are sufficiently powerful and configurable we may be able to synthesize efficient code for a particular market niche. Configuration of transformations by mark models will be essential if we are to retain the ability for domain experts to achieve subtle optimisations but avoid the need for the detailed low level manual coding and optimisations of too many current developments.

UML, MOF and OCL already provide a useful although imperfect ability to define abstract models for systems. Transformation technology has only just begun to rise to the challenge of MDA with models for transformation between models.

In this paper we consider how meta-model based transformations can exploit MOF meta-modelling to avoid many errors that arise with more traditional programming approaches. We therefore first review the nature of errors in existing technologies, and then introduce the UMLX notation highlighting similarities between UMLX and many other notations, but also reviewing some of the more significant differences between notations. We then take an

example from the QVT FAS [12] to demonstrate how a tool with strong type checking could avoid an unfortunate error in that example.

1.1 Errors

Errors are an unavoidable part of any human activity and their speedy elimination is very desirable. Programming errors may be detected at edit-time, compile-time, link/load-time or run-time. An edit-time error is detected almost instantly as soon as some construct is able to be analysed. A compile-time error is only detected once some program module is presented for analysis. Link or load-time errors must await until the possibly complete program is available for analysis. Finally a run-time error is not detected until a test execution happens to take some path that activates the error and an alert observer identifies the inappropriate behaviour. Delay in detection of errors obviously enables the errors to remain unresolved for longer. At the system level it has been estimated that delayed detection of a problem through each of the analysis, development, production and maintenance phases may incur a ten-fold increase in cost for each phase of delay.

In a conventional language such as C, type checking is able to find some programming problems, but the limited degree of checking of pointers allows many errors to go undetected. Limitations of the C syntax make provision of checking earlier than compile-time quite difficult.

The powerful class definitions of a language such as C++ can be viewed as an opportunity to define a customised compiler, and so user-defined types and the C++ language extensions can provide a much stronger checking environment. However syntax complexities make type checking earlier than compile-time hard, and the macro-like characteristics of templates make even compile-time checking difficult.

A more modern language, such as Java, with a disciplined syntax, can support incremental type checking as evidenced by the Eclipse JDT environment, where a concurrent thread updates the editor screen with error markers within a second or two. Integration of program understanding at edit time enables a wide variety of navigation, browsing, suggestion and correction facilities to be provided. This helpful but not overpowering checking contributes greatly to programmer productivity and satisfaction.

An extensible language such as XML supports only the most rudimentary syntax checking, and so an XSD schema is essential to impose some form of discipline. This supports a much improved editing environment, but only for a textual representation that should not be regularly exposed for direct human manipulation.

An extensible programming language such as XSLT provides much needed model transformation capabilities but subject to the dual handicaps of an unreadable textual representation and an uncheckable expression format. It is only with the advent of schema-awareness that XSLT can provide some checks on validity, but it is difficult to generate errors for the many inadvertent but plausible expressions that select nothing. XSD schemas unfortunately lack the relevance and precision of MOF models, potentially augmented by OCL constraints.

1.2 Error Avoidance

Recent tooling innovations with incremental compilation have succeeded in accelerating detection of many compile-time errors so that they are identified at edit-time. This is a major benefit, but we can do better, by making as many errors as possible impossible; an error that cannot be made needs no detection, diagnosis or correction. Thus few high level language programmers worry about the subtleties of stack or condition code corruption that provide so much entertainment for assembler programmers.

Certain classes of errors can be eliminated completely by use of a more powerful language or environment in which the particular error is impossible. Unfortunately this more powerful context is often more restrictive and less efficient. Consequently the more powerful context may be unattractive if not unacceptable for some applications. Provision of this context requires concepts that are more closely aligned to the programming intent if the benefits are to outweigh the restrictions.

Strong type systems have been particularly successful in this respect by allowing the compiler to be extended to support user-defined concepts. An MDA meta-model can be viewed as a definition of a constrained group of related types and so a natural evolution of type systems that enables programming tools to evolve to better align with a modelled domain and to prohibit programming statements that are in conflict with that domain. Initially this higher level of abstraction will no doubt result in even larger, even slower and even less efficient code. However, conversion of highly abstract models to concrete code through a progression of many transformations steered by a mark model provides an opportunity for re-usable transformations. These can gradually improve and perhaps overtake what is practically achievable by manual optimisations.

1.3 Transformation Notations

The basic concepts of transformations, rules and patterns of objects appear to be common to all transformation notations; a transformation comprises a number of rules[1] that each define a relationship between a matched pattern of objects on the input side[2] of the rule and a corresponding pattern of objects on the output side. Cosmetic but ergonomically important syntactical differences arise with respect to textual constructs or graphical presentation. More significant semantic differences occur in the permitted complexity of a pattern and the context in which a rule may be applied, although these differences have reduced as notations acquire more widespread use. Fundamental philosophical differences may be found in the definition of mappings from input to output patterns.

1.3.1 Semantic Differences

Early expositions of notations often support only patterns that match a single model element to each pattern element, but necessarily evolve, as QVT[12] has, to provide some support for a match to the collection of all elements that lie at the end of some association. UMLX extends UML multiplicities to support sub-sets of those collections too, enabling a pattern to be applied pair-wise or even combinatorially. This extension naturally supports a zero-wise match

[1] relations in QVT and UMLX
[2] domain in QVT and UMLX

that can only be true for an absence of elements; an empty collection. QVT introduces a {not} keyword instead.

The first definition of UMLX [18] required the context of all input and output rule elements to be bound by the invoking context. UMLX [17] now follows QVTrelation in allowing nested rules (relations) to be bound and top level rules to be unbound. Conversely, ATL [9] originally required all rules to apply universally. ATL now supports lazy and invoked rules that can be bound to specific contexts.

How to handle a pattern of objects that satisfies more than one rule is another source of semantic variation. ATL requires that no multiple match exist, whereas QVT applies to each distinct match of each rule. In order to generalise to matching of sub-sets, UMLX defines that each distinct pattern match should involve the largest possible number of elements and that each distinct pattern match of each rule is applied exactly once.

Any attempt to implement the UML to RDBMS example soon reveals the need to synchronise the behaviour of multiple matches across multiple rules. This ensures that the inputs are related to equivalent outputs in each match.

1.3.2 Philosophy Differences

Fujaba [8], GReAT [2], MOLA [10], QVToperational [12] and VIATRA2 [3] use an imperative semantics to define the mapping from input to outputs, while AGG [5], ATL [9], Gmorph [14], MT [15], MTF [11], QVTrelation [12], Tefkat [7] and UMLX use a primarily declarative semantics. Of these, Fujaba, GReAT, MOLA, VIATRA2, AGG, Gmorph, QVTrelation and UMLX provide graphical notations.

Imperative transformations are inherently unidirectional, but declarative transformations may be bi- or even multi-directional. ATL and Tefkat are practical transformation notations and so restrict their aspiration to unidirectional behaviour. QVTrelation and UMLX are more abstract, requiring, that amongst other things, their (potentially) multi-directional properties are resolved to create a unidirectional transformation once the required transformation direction has been established from its invocation context.

Graphical notations for imperative or procedural notations have not been particularly successful when applied in the form of a Program Flow Chart, Schlaer-Mellor Data Flow Diagram [13], or SDL. It is too early to tell whether modern tools and the more abstract meta-model based constructs used for model transformations are sufficient to allow Fujaba, GReAT, MOLA or VIATRA2 to overcome the limitations of earlier attempts at imperative graphics. It is also too early to tell whether the more consistently declarative exposition available in a graphical notation can enable AGG, Gmorph, QVTrelation or UMLX to encourage development of declarative rather than imperative transformations. It may perhaps take a long time for programmers to change their procedural and/or textual habits.

1.3.3 Graphical Differences

The semantics of UMLX are intentionally very similar to QVTrelation. The differences lie primarily in the provision of more powerful matching and mapping capabilities that suit a graphical exposition. UMLX graphics has only limited similarities to the proposed graphical notation for QVTrelation.

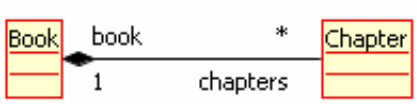

Figure 1 Example of basic UML/MOF meta-modelling constructs

The basic UML class diagram notation for meta-models shown in Figure 1 is very familiar and has formed the basis for some graphical notations. This has led to some confusion since the notation defines relationships between classes rather than between instances or matches to those classes.

UMLX re-uses UML concepts in so far as possible, and so the UML (multi-)object instance notation is extended as shown in Figure 2. Each node in the pattern is a class variable, typed by a class from the meta-model, to which (multiple) elements are assigned from the matched model. Each edge in the pattern is a constraint typed by an association from the meta-model. Both input and output patterns represent statements that are maximally true whenever the rule applies and each rule applies as often as possible.

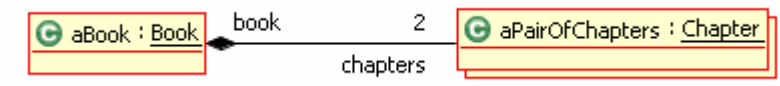

Figure 2 Example of UML object instance constructs (in UMLX)

`aBook` is a Class Variable of type `Book`, to which a particular book must be bound when the pattern matches. The relationship between `aBook` and `aPairOfChapters` imposes a Constraint upon the Class Variables at its ends; the Constraint is based upon the `Book` to `Chapter` association. Since a Class Variable is clearly different to a Class, there is no problem when a more complex pattern involves `anotherBook`. The multi-object notation is used for collections, and so a successful match of the pattern will result in a binding of a collection of chapters to `aPairOfChapters`, and since the pattern multiplicity is `2`, the matching collection must contain precisely two chapters. The pattern is therefore maximally satisfied for all pair-wise combinations of two chapters from books with two or more chapters.

Graphical Style

In the QVT FAS, 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.

Fujaba and MOLA underline both class and instance name but also omit decorations.

Gmorph and GRE [1] show line decorations in a similar way to UMLX, but Gmorph underlines both instance and class name while GRE uses a stereotype notation[3] for the class name.

AGG 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] text between angle brackets

Mapping Semantics

Graphical Transformation notations based on imperative principles combine a declarative exposition of patterns with a procedural exposition of the sequencing and control flow between the patterns. The graphical exposition requires little additional assistance from textual annotations and has some similarities to UML activity diagrams.

With the exception of UMLX, Graphical Transformation notations based on declarative principles rely on non-graphical annotations to define the mapping between input and output patterns.

UMLX appears to be the only Graphical Transformation notation that uses graphics for both a declarative exposition of the pattern and a declarative exposition of the mapping. Additional text is only required when, in the interest of clarity, it is better to hide unnecessarily detailed annotations.

2 Model Instantiation

The MOF QVT FAS [12] defines the QVTrelation language in terms of the QVTcore language using a QVTrelation transformation to define the semantics.

The first half page of a 16 page transformation defines the `RelationToTraceclass` Relation. Its exposition shown in Figure 3 no doubt represents the best endeavour of its author, who clearly lacked appropriate tool support. The example contains a readily diagnosed type error that is easily made when its textual representation has no inherent correlation with its meta-model. The error makes it impossible to enter the example into UMLX, which imposes compliance with the meta-model.

```
// Rule 1: Corresponding to each relation there exists a trace class in core.
// The trace class contains a property corresponding to each object node in the
// pattern of each domain of the relation.
//
relation RelationToTraceclass {
  checkonly domain relations r:Relation {
    name = rn,
    domain = rd:RelationDomain {
      pattern = t:ObjectTemplateExp {
        bindsTo = tv:Variable {
          name = vn,
          type = c:Class {}
        }
      }
    }
  }
  enforce domain core rc:Class {
    name = 'T'+rn,
    ownedAttribute = a:Property {
      name = vn,
      type = c
    }
  }
  where {
    SubTemplateToTraceClassProps(t, rc);
  }
}
```

Figure 3 `RelationToTraceclass` example from MOF QVT FAS

After correction of the error, we get the graphical equivalent shown in Figure 4. The solid lines on left and right hand domains denote the pattern that is matched by each set of related left hand and right hand side elements.

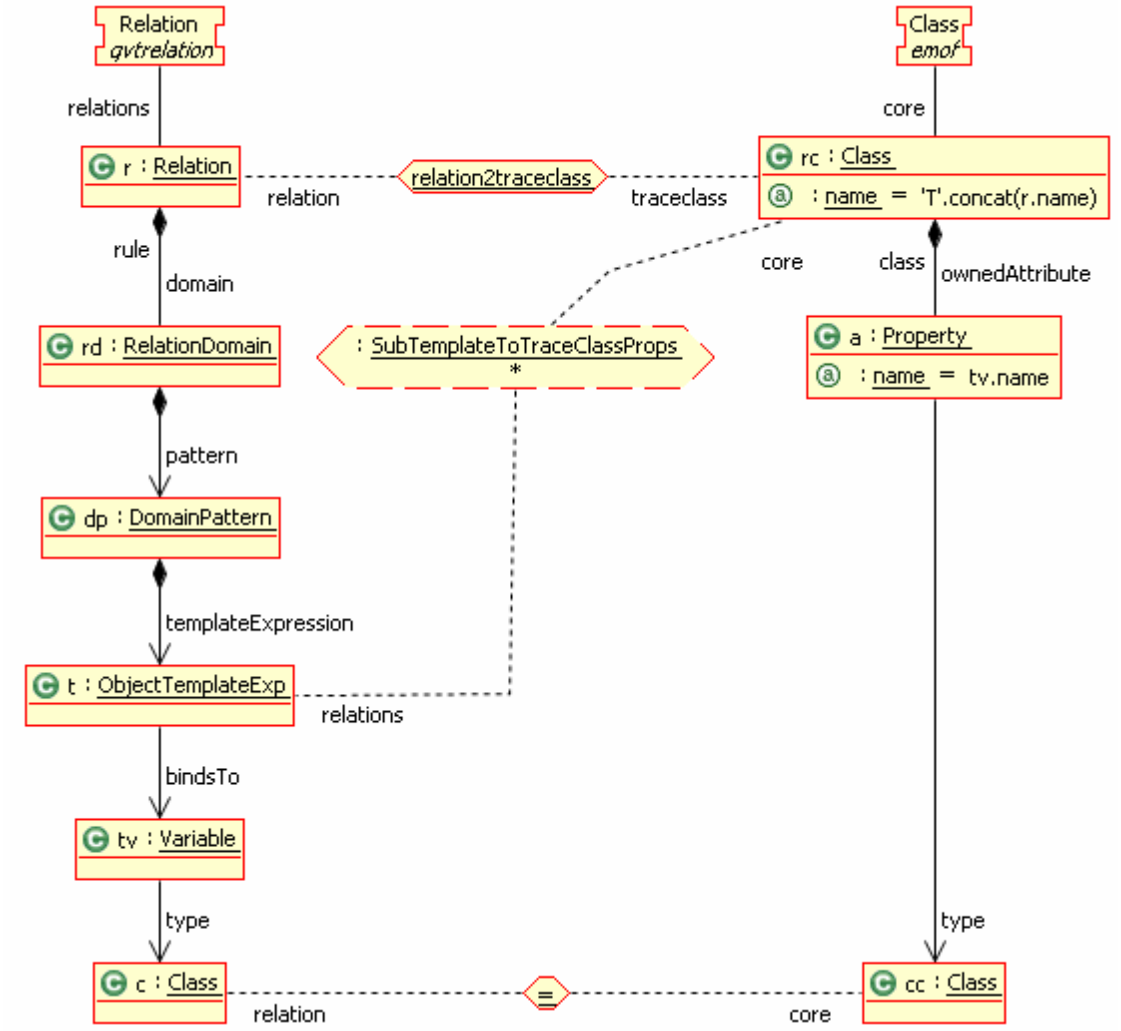

Figure 4 UMLX version of `RelationToTraceclass` Relation

This diagram is drawn primarily by dragging and dropping elements from an Outline view of the QVT meta-model, with remaining elements selected from a Drawing Palette.

The Outline View at the right of Figure 5 shows a tree view of the graphical UMLX model comprising 4 diagram sheets, a temporarily read-write locked UMLX Ecore model for the transformation and six read-only Ecore models one for each package of the QVT meta-model. The `RelationDomain` class of the `qvtrelation` package is open showing some of the model elements that could be dropped into the transformation pattern; dropping a Class generates a Class Variable in a pattern, dropping an Association generates a Constraint between two appropriately typed Class Variables; these are automatically created if not present. Since Class Variables and inter-Class Variable constraints are direct instantiations of

Classes or Associations in the meta-model, they are able to adopt the visual style of their instantiated element; the diamond, arrow and label decorations on the Constraint between `RelationDomain` and `DomainPattern` are therefore drawn automatically and change automatically as the underlying meta-model is changed.

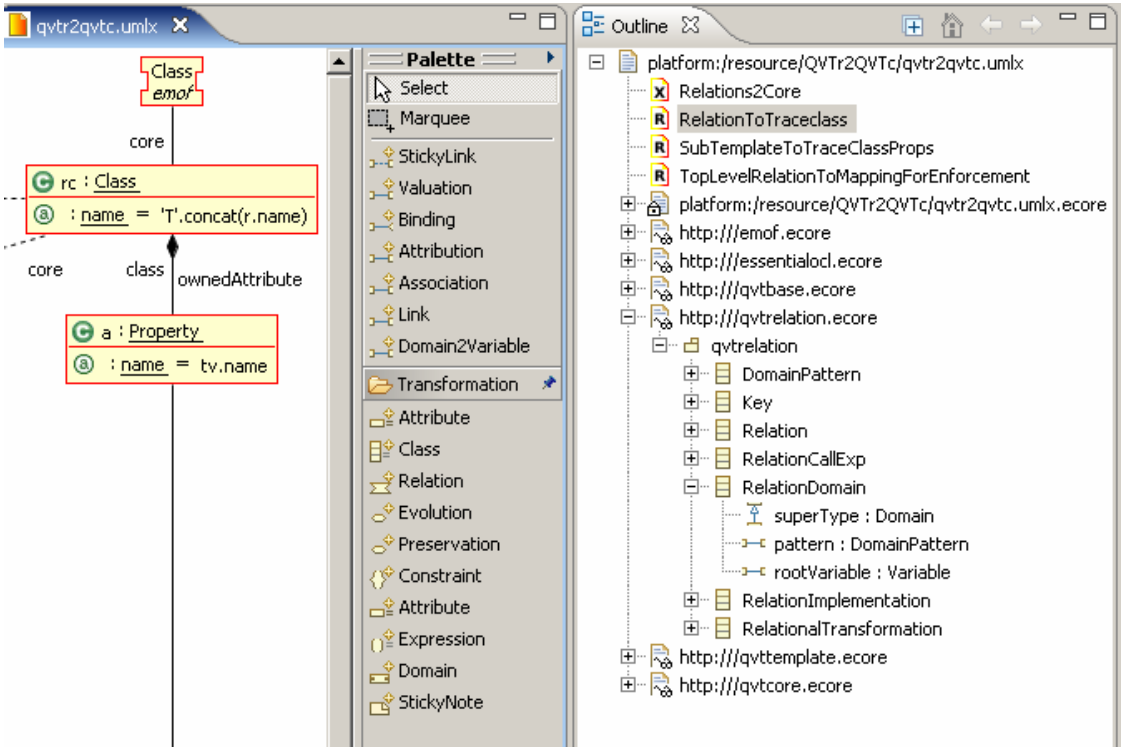

Figure 5 UMLX Palette and Outline

The meta-model in the Outline shows that a `RelationDomain` has a `pattern` as required by the QVT specification, but that a `pattern` must be a `DomainPattern`, not an `ObjectTemplateExp` as required in Figure 3. Entering the example with UMLX reveals the problem and the relatively simple bug fix that introduces the missing `DomainPattern` between the `RelationDomain` and the `ObjectTemplateExp`.

2.1 Mappings

The UMLX version of the example demonstrates the three UMLX mapping operators, each of which exploits instantiation to reduce opportunities for errors.

Graph Transformations [6] use the basic concepts of Add, Delete and Keep to define the mapping between input and output elements. UMLX generalises the unidirectional Add and Delete operators for single objects to a multi-directional Evolution that exhibits traceability characteristics to support multi-objects, inheritance and synchronisation of multiple rules. UMLX generalises the Keep operator as a Preservation that supports a deep 'copy' of all nodes and edges within the sub-graph defined by composition relationships rooted at the preserved object.

2.1.1 Evolution

The instance of the `relation2traceclass` evolution at the top of Figure 4 defines the evolution of elements of the input domain(s) to elements of the output domain(s). When the example transformation is used in a left to right direction, it requires a trace `Class` to exist for each `Relation`.

Since the graphics instantiates rather than defines an evolution, the same evolution may be instantiated by another rule that requires synchronisation between input and output elements. Instantiation of the evolution must of course comply with the definition. Therefore all input and output elements of the same evolution must be compatibly typed, and of course the evolution from the same set of input objects necessarily yields the same set of output objects in all relations instantiating the same evolution. The example is a simple A to B conversion; the generalised definition applies to multiple inputs and outputs, each of which may be a collection of objects.

2.1.2 Invocation

A UMLX relation can be invoked, from another relation or from itself, by binding the input and output domains of the invoked relation to the matched objects associated with class variables in the invoking context. The example shows how the `relations` and `core` domains of the `SubTemplateToTraceClassProps` relation are bound to the `ObjectTemplateExp` and `Class` matches of the `RelationToTraceclass`. The example also shows the declaration of the `relations` and `core` domains of the `RelationToTraceclass` as matches for a `Relation` from the `qvtrelation` package, or for a `Class` from the `emof` package.

A relation may be invoked declaratively as a constraint that must also be satisfied *when*ever a matching pattern of input elements is identified. Since the invocation is unconditionally true for this usage, UMLX uses a lozenge with solid lines; solid lines are always true.

Alternatively, as in the example, a relation may be invoked procedurally so that the relation is invoked *where*ver a matching pattern of input elements is identified. Since the invocation is optionally true, UMLX uses a lozenge with dashed lines for this usage; dashed lines may be true.

2.1.3 Preservation

The mapping operator with an equality sign near the bottom of Figure 4 is a preservation operator indicating a deep equivalence (copy) between the composition trees rooted at the input and output elements. A preservation is also instantiated so that multiple instantiations of the same preservation of the same objects can synchronise rules. Alternatively instantiation of multiple preservations supports multiple copies of an input element.

3 Model Type Checking

We have shown how entry of a transformation model using a graphical notation can exploit and enforce the type system defined by the meta-model and so prevent entry of illegal statements.

Once a model has been successfully entered, it may be persisted using an XMI serialisation. If the serialisation uses XMI identifiers to link model to meta-model, the instantiation can survive many simple forms of refactoring, such as a change of name or reorganisation of a class hierarchy. Alternate forms of XML serialisation, that resolve references using XML node position or hierarchical XML names, are much less tolerant of refactoring.

More complex meta-model changes that represent a semantic rather than cosmetic change are of course not survivable but can be diagnosed within the editor.

Figure 6 shows the UMLX display for the earlier example, after simulating a typical design evolution. The QVT meta-model was modified to exclude the `DomainPattern` class and so allow the example to be entered as an apparently valid model. The modification was then removed from QVT meta-model simulating a typical design improvement. The consequence of this meta-model change shows how the incompatibility is brought to the users attention.

The Problems View identifies the problem and supports navigation to the erroneous design element. The Editor view shows an error marker on the `RelationDomain` to `ObjectTemplateExp` association, which is in error because the QVT meta-model requires the type of the `pattern` to be `DomainPattern`.

Note that this error cannot arise during design entry; a model can only be invalid with respect to its meta-model if the meta-model is changed after the model was entered.

4 Conclusion

We have discussed how an MDA based on progressive model transformations offers an improved programming environment with reduced opportunities for errors.

We have identified that a textual notation lacks inherent compliance with its meta-model and so requires sophisticated tool support for any form of error checking or even simple forms of refactoring.

In contrast we have shown how a graphical transformation can be closely coupled to its meta-model and so can provide insight through visual decorations, avoid some forms of design error, detect many others and readily support basic refactorings and model navigation.

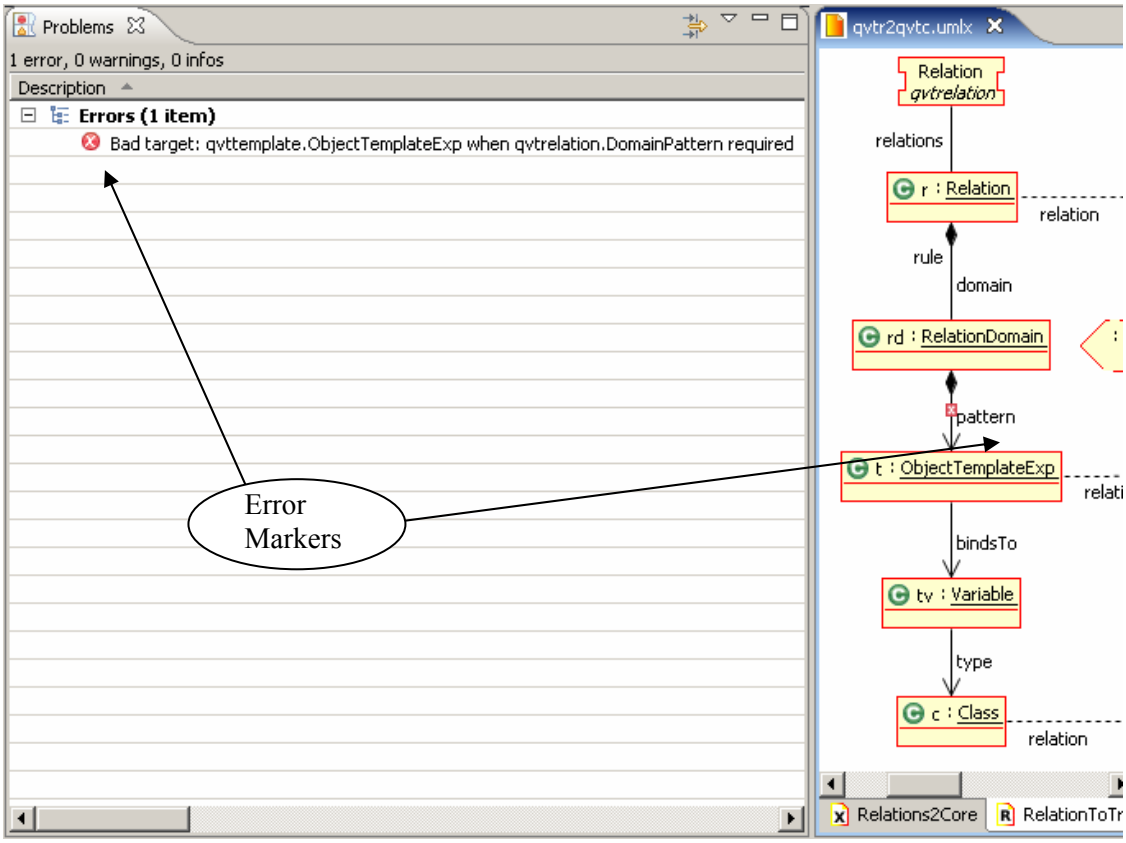

Figure 6 UMLX Views with errors

Readers may judge for themselves whether the graphical or textual presentation of the particular example in this paper is easier to grasp. It seems likely that simple or list-like transformations are better expressed textually whereas complex patterns may be better expressed graphically [16].

In order to combine the disciplines and advantages of the graphical approach with the apparent compactness of the textual, both textual and graphical notation should be realised as alternate views of the same underlying model, thereby allowing the user a free choice of the most appropriate view for their purpose.

5 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 MDA*, November 2002, 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", http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_0_0_2003_A_UML_base.pdf

[3] Balogh, A., Varró, D.: "Advanced Model Transformation Language Constructs in the VIATRA2 Framework", *ACM Symposium on Applied Computing --- Model Transformation Track*, SAC 2006.

http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2006/sac2006_vtcl.pdf

[4] Brooks, Frederick P.: "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4 (April 1987) pp. 10-19.

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

[6] 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.

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

[8] Fujaba, http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf

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

[10] Kalnins, A., Celms, E., Sostaks, A:. "Model Transformation Approach Based on MOLA". *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS/UML '2005, Montego Bay, Jamaica, October 2 -7, 2005, p. 25.
http://melnais.mii.lu.lv/audris/Model%20Transformation%20Approach%20Based%20on%20MOLA.pdf

[11] MTF, http://www.alphaworks.ibm.com/tech/mtf

[12] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification", OMG Final Adopted Specification, ptc/05-11-01, http://www.omg.org/docs/ptc/05-11-01.pdf

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

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

[15] Tratt, L.: "The MT model transformation language", *Proc. ACM Symposium on Applied Computing*, pages 1296-1303, April 2006
http://tratt.net/laurie/research/publications/papers/tratt__the_mt_model_transformation_language_sac.pdf

[16] Willink, E.: "On Challenges for a Graphical Transformation Notation and the UMLX Approach", *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques*, GT-VMT 2006, Vienna, March 2006
http://dev.eclipse.org/viewcvs/indextech.cgi/*checkout*/gmt-home/subprojects/UMLX/doc/GT-VMT2006/GTVMT2006.pdf

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

[18] Willink, E.: "UMLX : A Graphical Transformation Language for MDA", *OOPSLA 2003 Workshop on Generative Techniques in the context of MDA*, 2003
http://dev.eclipse.org/viewcvs/indextech.cgi/*checkout*/gmt-home/doc/umlx/Oopsla2003.pdf