Model Transformation – the Heart and Soul of Model-Driven Software Development

Shane Sendall and Wojtek Kozaczynski

Swiss Federal Institute of Technology in Lausanne (EPFL)
Software Engineering Laboratory
1015 Lausanne EPFL, Switzerland[1]
sendall@acm.org

Microsoft, Prescriptive Architecture Guidance Group
Redmond, WA, USA
wojtek@microsoft.com

The motivation behind model-driven software development is to move the focus of work from programming to solution modeling. The model-driven approach has a potential to increase development productivity and quality by describing important aspects of a solution with more human-friendly abstractions and by generating common application fragments with templates. For this vision to become reality, software development tools need to automate the many tasks of model construction and transformation, including construction and transformation of models that can be round-trip engineered into code. In this article, we briefly examine different approaches to model transformation and offer recommendations on the desirable characteristics of a language for describing model transformations. In doing so, we are hoping to offer a measuring stick for judging the quality of future model transformation technologies.

One of the best ways to combat complexity of software development is through the use of abstraction, problem decomposition, and separation of concerns. The practice of software modeling has become a major way of implementing these principles. Model-driven approaches to systems development move the focus from third-generation programming language (3GL) code to models (in particular models expressed in UML and its profiles). The objective of model-driven development is to increase productivity and reduce time-to-market by enabling development at a higher level of abstraction and by using concepts closer to the problem domain at hand, rather than the ones offered by programming languages. The key challenge of model-driven development is in transforming these higher-level models to so-called platform-specific models that can be used to generate code.

Over the last few years, the software development industry has gone through the process of standardizing visual modeling notations. The Unified Modeling Language

[1] From August 2003, Shane's address will be: Software Modeling and Verification Lab., University of Geneva, CH-1211 Geneva 4, Switzerland.

(UML) [Omg03a] is the product of this effort, and it unifies scores of notations that were proposed in the '80s and '90s. The language has gained significant industry support and became an Object Management Group (OMG) standard in 1997. The majority of software modeling techniques and approaches use UML, and the language and its profiles have been associated with the model-driven development vision.

UML gives numerous options to developers for specifying software systems. A UML model can graphically depict the structure and/or behavior of the system under discussion from a certain viewpoint and at a certain level of abstraction. This is desirable as one can typically better manage the complexities of a system description through the use of multiple models, where each captures a different aspect of the solution. Models can also be refined and decomposed into other models. Thus, models can be utilized not only in a horizontal manner (to describe different system aspects), but also in a vertical manner (to be refined from higher to lower levels of abstraction).

Working with multiple, interrelated models that describe a software system require significant effort to ensure their overall consistency. It follows that automating the task of model consistency checking and synchronization would greatly improve the productivity of developers and the quality of the models. In addition to vertical and horizontal model synchronization, the burden of other activities, like the ones listed below, could be significantly reduced through automation:

- Refinement: The development of an application can be logically viewed as combination of many steps taking one from requirements to realization. For example, a specification of a component can be refined to include a state machine describing state transitions resulting from receiving messages on its ports.
- Reverse Engineering Models: Going from concrete models to more abstract ones can be useful for modelers who wish to work and/or communicate at a higher level of abstraction. For example, complex interactions between many components may be abstracted into aggregate interactions between component layers.
- Generating New Views: The generation of different views from existing models can be useful for concentrating on a particular concern of the system where non-pertinent information (to the concern) is filtered out. For example, components that do not use remote communication can be filtered out as non-distributable components.
- Applying Software Patterns: The need to apply architectural and design patterns as well as class-level idioms arises often during software development. For example, one may want to change local component communication to remote communication which will require coordinated changes to all components that depend on it. Note that the application of a pattern in many cases is a refinement step, and thus can be seen as overlapping with the first point.
- Refactoring Models: The accumulation of changes to a model can make it complex and difficult to maintain. A way to improve the form of a model is to refactor it. Refactoring a model involves changing its structure while preserving its behavior. Refactoration is usually achieved by the application of refactoring patterns and has been given a separate bullet here because it has achieved quite some interest recently. An example of refactoring could be the splitting of parts of a complex component into separate components.

Many of these activities can be performed as automated processes, which take one or more source models as input and produce one or more target models as output, following a set of transformation rules. We refer to this process as *model transformation*.

For the model-driven software development vision to become reality, tools must be able to support the automation of model transformations. Development tools should not only offer the possibility of applying predefined model transformations on demand, but should also offer a language that allows (advanced) users to define their own model transformations and then execute them on demand. Beyond transformation execution automation, it would also be desirable that tools could make suggestions as to which model transformations could be appropriately applied in a given context, but this aspect is out of the scope of this article.

In this article, we analyze current approaches to model transformation, but we concentrate on desirable characteristics of a model transformation language. Such a language can be used by modeling and design tools to automate tasks like patterns application, refinement or refactoring. Tools that implemented a language with the described characteristics would not only support the Model-Driven Development paradigm, but more importantly, could significantly improve development productivity and quality.

Classifying Approaches to Model Transformation

Performing a model transformation, taking one or more models as input and producing one or more models as output, requires a clear understanding of the abstract syntax and the semantics of both the source and target. A common technique for defining the abstract syntax of models and the inter-relationships between model elements is meta-modeling. Practice has shown that for visual modeling languages there are a number of advantages in basing a tool's implementation upon the meta-model of the language. A number of tools exist which allow one to define a domain-specific visual language by the specification of a meta-model, e.g., [Dome, GME, MetaEdit+, ParadigmP]. UML is also specified in terms of a meta-model, which is implemented (at least partially) by a large number of tools, e.g., [Objecteering, RationalRose, RationalXDE, Together].

These tools, in general, offer the user one of three different architectural approaches for defining transformations[2]: the direct model manipulation approach, the intermediate representation approach, or the transformation language support approach.

[2] This classification was inspired by [Wksp-DSVL], which classifies approaches that generate code from models.

- Direct Model Manipulation (sometimes referred to as Pull) – the tools offer users access to an internal model representation and the ability to manipulate the representation using a set of procedural APIs.
- Intermediate Representation – the tool can export the model in a standard form, typically XML. An external tool can then take the exported model and transform it.
- Transformation Language Support – the tool offers a language that provides a set of constructs or mechanisms for explicitly expressing, composing and applying transformations.

A tool that offers one (or more) of these three approaches provides a means to define model transformations. In what follows, we highlight some of the advantages and limitations of the different approaches.

An advantage of the direct manipulation approach is that the language used to access and manipulate the exposed APIs is either a common language like VB or Java or a proprietary variant of a common language, and the developers need little or no extra training to write transformations. Furthermore, developers are generally more comfortable with encoding complicated (transformation) algorithms in procedural languages. Examples are Rational Rose [RationalRose] that offers a version of VB with a set of APIs to manipulate models and Rational XDE [RationalXDE] that exposes an extensive set of APIs to its model server that can be used from Java, VB or C#. A disadvantage of using this approach is that the API usually restricts the kind of transformations that can be performed. Also, since the programming languages are "general-purpose", they lack suitable high-level abstractions for specifying transformations. As a consequence, encoding transformations can be time-consuming, cumbersome, and the transformation algorithms may be difficult to maintain. One proposal that promises to raise the level of abstraction of operations on UML models is UML's action language [Omg03a]. The language has been proposed as a way to procedurally define UML transformations [MB02, SPH+01] and is a special-purpose language for manipulating UML models. However, due to its "general-purpose" context, the UML action language still suffers, albeit less chronically, from a lack of high-level abstractions for dealing with model transformations like, for example, transformation composition.

With respect to the intermediate representation approach, many UML tools offer the facility to export and import models to/from XMI. XMI is an XML-based standard for interchange of UML models [Omg03a]. Because a model is externalized into XML, it is possible to use existing XML tools, such as XSLT [XSLT], to perform model transformations. Some researchers have also proposed the use of the XMI.difference clause defined by XMI [Wag01], however this latter type of approach offers significantly less expressive power compared to XSLT-based approaches. Even though XSLT was defined for the specific purpose of describing transformations, it is nevertheless tightly coupled to the XML that it manipulates. As a consequence, it requires experience and considerable effort to define even simple model transformations in XSLT. To address this problem, MTrans proposes a language that is placed on top of XSLT to describe model transformations [PZB02], where XSLT is

generated from an MTrans description. A drawback of MTrans is that even though the idea is indeed very promising, the proposed language possesses a number of the idiosyncrasies of XSLT, e.g., a restrictive functional style. Another disadvantage of the intermediate representation approach is that transformations are performed in batch mode. There are two important consequences of that. One is that transformations are hard to perform in an interactive dialog with the user. The other one is that the tool still needs to reactively manage the synchronization between models after changes. For example, a long and complex transformation performed outside of the tools may be rejected due to the violation of cross-model integrity constraints.

Transformation language support, as the name suggests, provides a domain-specific language for describing model transformations and, by consequence, offers the most potential of the three approaches. Within this context, there are many languages that can be used to specify and execute model transformations, some of which offer visual constructs. These languages are either declarative, procedural, or a combination of both. Below are a few examples.

The work described in [Mil02] proposes a graphical language for describing model transformations that is principally procedural in nature but also offers some declarative features. The proposed approach offers a UML object diagram as notation for developing the mapping specification. The notation has been extended, using UML's stereotype extensibility mechanism, with constructs for conditional, repetitive, parameterized, and polymorphic model element creation. These concepts, theoretically, allow one to generate any kind of model as a result of transformation. The use of a graphical notation for defining the mapping specification is likely to make the approach more accessible to users (especially when you compare it to the equivalent C++ code), even though the graphical form makes heavy use of stereotypes and uses common UML elements, such as, packages, in ways that are not typically seen in UML-based languages. The approach is supported by a tool that generates C++ code from the mapping specification. A limitation of the proposed approach is its underlying assumption that the selection of source model elements for the transformation can be easily expressed in a general-purpose programming language, i.e., C++. If one were faced with complex selection criteria, it would be very likely that these selection conditions would become complex and hard to maintain. In fact, it would be at least useful to offer a language that is tailored for such a purpose, such as, UML's Object Constraint Language (OCL) [Omg03a, WK98].

A commercial example of a specialized transformation language is the Rational XDE's pattern mechanism [RationalXDE]. XDE transformations are defined as model templates called Patterns. A pattern may contain parameters and also may contain arbitrary procedural code written in Java, VB or C#, which is invoked by a set of pre-defined call-backs. At the time of application, the pattern engine binds pattern parameters with arguments either automatically or assisted by the user and expands the pattern into the target model. If a procedure is associated with a callback, the pattern engine passes a handle of the model to the procedure, which effectively means

that the user can make arbitrary "manual" changes. The key drawback of the XDE's pattern engine is that it provides a limited capability to compose patterns.

Another technique is to treat UML models as graphs. Much work has been performed on graph grammars and graph transformation systems. Graph transformations are realized by the application of transformation rules, which are rewriting rules for graphs. A transformation rule consists of a graph to match, commonly referred to as LHS, and a replacement graph, commonly referred to as RHS. If a match is found for the LHS graph, then the rule is fired, which results in the matched sub-graph of the graph under transformation being replaced by the RHS graph. The PROgrammed GRaph REplacement System (PROGRES) [SWZ97] exemplified the means not only to specify transformation rules but also to define the sequencing of these rules (described using imperative constructs). This feature of PROGRES sets it apart from many of the other graph transformation approaches.

Unfortunately, PROGRES provides no direct support for UML. The Fujaba environment, a specialized successor to PROGRES, provides round-trip engineering support for UML and Java [KNN+00]. Unfortunately, it is not clear how Fujaba could be generalized for UML-to-UML transformations, as it uses graph transformations for the purpose of visual programming.

Another graph transformation system for domain-specific model transformations is the Graph Rewriting and Transformation Language (GReAT for short) [AKS03]. Similarly to PROGRES, it separates the language for describing transformation rules from the language for describing rule ordering. In GReAT, metamodels for the source and target models are used to establish the vocabulary of the LHS and RHS and to ensure that the transformation produces a well-formed target model. Surprisingly, GReAT's rule language defines LHS, RHS and a set of explicit transformation actions in a single graph. Even though it is not ambiguous to the tool's interpreter, it unfortunately makes rule comprehension more difficult. GReAT's rule composition language has a visual form that resembles a circuit diagram. It offers a number of operators for sequencing rules, non-deterministic ordering of rules, rule composition, recursion, and conditional branching. Even though the rule composition language offers quite some expressive power, using it in the specification of complex transformation composition strategies and algorithmic heavy analyses would require quite some training in the language. In general, it is very difficult to use a visual language to write complicated algorithms[3].

Another technique exemplified by the transformation framework based on Maude, a logic-based programming language [CDE+01], is described in [Whi02]. Maude code consists of a set of equations and rewrite rules. The Maude execution engine applies these rules to transform a given term. The UML abstract syntax is provided to the Maude engine as a set of theories, and from this, transformation rules can be defined as rewrite rules, which work in a similar way to the graph transformation approach. Although the work is a good step in the right direction, writing rules in a logic language like Maude is not simple. For example, the way that parameters are

[3] This is confirmed by the lack of popular general-purpose programming languages that have a visual notation.

bound to source or target model elements is not intuitive to inexperienced users, as variables can become any element necessary to satisfy the rule, existing or otherwise. Consequently, readability and understandability of the description may be an initial hurdle if a logic programming language is used.

The work described in [SPG+03] offers a pragmatic way to address some of the shortcomings of graph-based transformation languages. It offers a visual language similar to GReAT, however, it uses the philosophy that many of the complicated algorithms in model transformation are easier to realize in a general-purpose programming language such as C# and Java. The proposed approach places the model transformation language on top of the programming language, in much the same way that an integrated development environment (IDE) such as Visual Studio, offers a visual GUI builder on top of the application code. This aspect of VMT has both advantages and disadvantages. An advantage is the availability of a set of abstractions that are fine-tuned for UML-to-UML transformations and the accessibility and expressive power of a general-purpose programming language. A disadvantage is that the user is required to work in multiple languages that have different levels of abstractions. VMT also has features beyond those present in GReAT. VMT's language for rule specification offers not only a metamodel-level view of the LHS and RHS, but also a model-level view, which makes it easier for users that are not familiar with the details of the UML metamodel. The rule specification language also makes a clear separation between the LHS and RHS, which means rule comprehension easier.

Desirable Characteristics of a Model Transformation Language

Building upon the discussion in the previous section, in this section, we propose a set of characteristics that we believe are desirable for a model transformation language to possess in general.

The languages that we surveyed in the previous section vary from principally visual notations to text only notations, from highly declarative to fully imperative, and from containing a small set of general language constructs to a large number of specialized language constructs. What then is the optimal kind of language for expressing model transformations?

If a language is to have general utility and acceptance, then it should have full expressive power for the chosen purpose and it should be implementable in an efficient way. Even if we were to limit the language to UML-to-UML transformation, we would still require a fully expressive language. This is because UML can be used in an almost unlimited number of ways[4], and it seems unlikely that we could predict the kinds of analyses and strategies that would be needed for UML-to-UML

[4] To gauge how many ways UML can be used, one only needs to contemplate the number of potential domains of usage and the number of different possible purposes with which UML can be used in each domain.

transformations in general. As a consequence, the transformation language would probably need to be Turing-complete.

Expressive power is only one important aspect of a transformation language. Usability is another equally important aspect. The usability of a language is a difficult subject to conquer because it depends not only on the purpose of the language but also on the preferences and backgrounds of its users, which is subjective by definition.

There are a number of factors that need to be addressed and balanced in a language. It should be easy-to-understand, yet precise and unambiguous. It should be concise and easy-to-modify, yet complete.

A declarative language offers an implicit interpretation such that one can take advantage of a set of underlying mechanisms to formulate the desired specification. For example, in the graph transformation approach, the algorithm for LHS graph matching is implicit and does not need to be expressed as part of the specification. On the other hand, an imperative language makes every step of the algorithm explicit. For example, procedural languages are imperative and they use procedures as abstraction mechanisms to encapsulate sets of instructions. As a consequence of the underlying and implicit mechanisms, a declarative language is typically more concise than a comparable imperative language. Nevertheless, there is a trade-off between conciseness and comprehension, where, if a specification has too many implicit and complicated concepts, it may be more difficult to understand than a more explicit, yet verbose, specification. As such, the key to the design of a transformation language is a set of key abstractions for transformation that are intuitive and cover the largest possible range of situations.

Many of the rules for mapping source model elements to target model elements can be made implicit and can be defined in a similar manner to the way that people communicate. As such, a declarative language can facilitate this aspect. For example, the intuitive interpretation of a certain schema may imply a depth-first traversal of the specification hierarchy. In this case, it would be desirable to make this implicit in the language. Nevertheless, in many of the approaches that we surveyed in previous section, imperative operators are commonly used in transformation composition, because this aspect of transformation description is more suited to an imperative interpretation.

The accessibility and acceptance of a language also depends on its form. One of the appealing features of UML is that it uses a graphical form. Graphical representations of models have proved popular because there are perceived cognitive gains compared to fully textual representations. In the context of a transformation language, specifying the structure of the input selection of a transformation using visual means is an appealing prospect. In any case, a graphical description is best complemented with textual parts, because in certain situations the use of text in the description is both more concise and easier to comprehend than an equivalent graphical representation.

With a large repository of model transformation descriptions at ones disposal, it follows that it may be desirable to combine existing transformations to build new, composite ones, since it is sometimes easier to compose components rather than build something from basic particles. Furthermore, in some cases it may be easier to build a transformation piecemeal by describing parts of the transformation first and then bringing together the parts to form the whole. Most of the approaches that we reviewed in the previous section offered language support for transformation composition and reuse.

Assuming that the transformation is correctly specified by the author and correctly interpreted by the tool, we would usually expect the transformation to produce a meaningful result. However, a transformation is typically only meaningfully applied against certain configurations of models. Thus, it would be desirable in many cases to describe the condition under which the transformation produces a meaningful result, which can then be enforced at execution time.

The following statement summarizes the desirable and recommended characteristics for a model transformation.

It is recommendable for a model transformation language which supports model-driven software development to:

1. *be executable;*
2. *be implementable in an efficient way;*
3. *be fully expressive, yet unambiguous, for transformations that modify existing models (add, change or delete model elements) as well as create completely new models;*
4. *facilitate developer productivity with precise, concise and clear descriptions:*
    - *the language should clearly differentiate the description of the source model selection rules from the rules for producing the target model;*
    - *the language should offer graphical constructs in the cases that the concepts represented are more concise and intuitive in graphical form compared to a textual one;*
    - *the language should be declarative by making implicit any concepts or mechanisms that can be intuitively interpreted from the context;*
5. *provide a means to combine transformations to form composite ones, offering at least operators for sequencing, conditional selection and repetition of transformations; and*
6. *provide a means to define the conditions under which the transformation is allowed to execute.*

Standardizing Model Transformations

Despite its poor initial definition, the concepts of the OMG's Model Driven Architecture (MDA for short) [OMG01a] are gaining interest. MDA is a framework for model-driven software development that defines three steps for going from high-level design to software realization [KWB03]:
Step 1: A model of the software system is constructed that is independent of any implementation technology, e.g. independent of J2EE, .NET, etc. This type of model is referred to as a Platform Independent Model (PIM).
Step 2: A PIM is transformed into one or more Platform Specific Models (PSMs), using a particular mapping strategy. A PSM specifies a system using implementation constructs that are available in one specific implementation technology, e.g., .NET platform, etc.
Step 3: The PSMs are transformed into code.

It is clear that to realize the MDA vision, one needs to be able to describe the transformation between PIM and different PSMs and then have tools transform the PIM based upon the description provided to it. Hence, we believe that MDA and its related activities will be the main stimulus for standardizing model transformation languages and there are early examples of that.

Even though the details of MDA are still being refined, there are already a number of tools that claim support for the MDA framework, e.g., [ArcStyler, OptimalJ].

Also, the OMG's Common Warehouse Metamodel (CWM) Specification [Omg01b] defines a generic way to describe white-box and black-box transformations (see chapter 13 of the specification). In CWM, black-box transformations associate source and target elements without describing how one is obtained from the other. The white-box transformations describe the fine-grained links between source and target elements by way of an explicit element called a Transformation, which is associated to another element called a ProcedureExpresssion. A ProcedureExpression defines the transformation operation; it can be expressed in any language capable of taking the source element and producing the target element(s). Thus, CWM does not offer any actual mechanisms or languages for performing the transformation; it is instead used to describe the existence of a mapping.

To fill this gap, OMG has posted a Request for Proposal called MOF 2.0 Query/Views/Transformations RFP [Omg02a], which has been answered by eight different initial submissions. The successful final submission will potentially add the much needed keystone to OMG's MDA vision, and it will need to address the characteristics that we identified in the previous section.

Conclusion

For software tools to become truly useful in aiding developers, they need to be able to automate the everyday tasks of users. With the potential impact of model-driven approaches on software development practices, tools will need to better automate the construction and evolution of software models. Currently, the best way to go about this goal is for tools to offer an executable model transformation language that allows one to automate model creation, development and maintenance activities. In this article, we proposed some desirable characteristics that this language should possess. In doing so, we offered a measuring stick for judging the quality of future model transformation technologies.

Acknowledgements

Shane's activities were sponsored by the FIDJI project n° MEN/IST/01/001, Luxembourg. Also, Shane would like to thank Olivier Biberstein and the FIDJI team at Luxembourg University of Applied Science.

References

[AKS03] A. Agrawal, G. Karsai, and F. Shi; "A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations". International Journal on Software and Systems Modeling, (Submitted), 2003.

[ArcStyler] ArcStyler Web Site, Interactive Objects Software, 2002. http://www.io-software.com/index_as.html

[CDE+01] M. Clavel, F. Durän, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesda; "Maude: Specification and Programming in Rewriting Logic". Theoretical Computer Science, 2001.

[Dome] DOmain Modeling Environment (DOME) Official Web Site, Honeywell International Inc., 2002. http://www.htc.honeywell.com/dome/

[GME] GME 2000 and MetaGME 2000, Institute for Software Integrated Systems, Vanderbilt University, 2002. Available at http://www.isis.vanderbilt.edu

[KNN+00] H. Köhler, U. Nickel, J. Niere, and A. Zündorf; "Integrating UML Diagrams for Production Control Systems". Proceeding of the International Conference on Software Engineering (ICSE200), Ireland, 2000.

[KWB03] A. Kleppe, J. Warmer, and W. Bast; "MDA Explained: the Practice and Promise of Model-Driven Architecture". Addison-Wesley, 2003.

[MetaEdit+] MetaEdit+ Workbench Web Site, MetaCase Consulting, 2002. http://www.metacase.com

[Mil02] D. Milicev; "Domain Mapping Using Extended UML Object Diagrams". IEEE Software, pp. 90-97, March/April 2002.

[Objecteering] Objecteering/UML Web Site, Objecteering Software SA, 2002. http://www.objecteering.com/us/index.php

[Omg01a] OMG Architecture Board ORMSC; "Model Driven Architecture (MDA)". July 9, 2001 (draft). http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01

[Omg01b] OMG CWM Partners; "Common Warehouse Metamodel (CWM) Specification", Feb. 2001 http://www.cwmforum.org/spec.htm

[Omg02a] OMG TC; "MOF 2.0 Query/Views/Transformations RFP", 2002. http://cgi.omg.org/cgi-bin/doc?ad/02-04-10

[Omg03a] OMG Unified Modeling Language Revision Task Force; "OMG Unified Modeling Language Specification". Version 1.5, March 2003. http://www.omg.org/technology/documents/formal/uml.htm

[OptimalJ] OptimalJ Web Site, Compuware Corporation, 2002. http://www.compuware.com/products/optimalj/

[ParadigmP] Paradigm Plus Web Site, Computer Associates (a product of the former Platinum Technology), 2002. http://ca.com/products

[PZB02] M. Peltier, F. Ziserman, and J. Bézivin; "On Levels of Model Transformation". In Proceeding of XML Europe 2000; Paris, France; 12-16 June 2000.

[RationalRose] Rational Rose Web Site, Rational Software Corporation, 2003. http://www.rational.com/products/rose/index.jsp

[RationalXDE] Rational XDE Web Site, Rational Software Corporation, 2003. http://www.rational.com/products/xde/index.jsp

[SPG+03] S. Sendall, G. Perrouin, N. Guelfi, and O. Biberstein; "Supporting Model-to-Model Transformations: The VMT Approach". Workshop on Model Driven Architecture: Foundations and Applications; Holland, 2003.

[SWZ97] A. Schürr, A. Winter and A. Zündorf; "The Progres Approach: Language and environment". In Chapter13 of G. Rozenberg, Handbook of graph grammars and computing by graph transformation: volume I foundations, World Scientific Publishing, 1997.

[Together] Together ControlCenter Web Site, TogetherSoft Corporation, 2002. http://www.togethersoft.com/products/controlcenter/index.jsp

[Wag01] A. Wagner; "A Pragmatic Approach to Rule-Based Transformations within UML using XMI.difference". WITUML: Workshop on Integration and Transformation of UML models (held at ETAPS 2001), 2001.

[Whi02] J. Whittle; "Transformations and Software Modeling Language: Automating Transformations in UML". Jézéquel, Hußmann & Cook (Eds.): Proceedings of UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany. Lecture Notes in Computer Science no. 2460, pp. 227-242, Springer, 2002.

[WK98] J. Warmer and A. Kleppe; "The Object Constraint Language: Precise Modeling With UML". Addison-Wesley 1998.

[Wksp-DSVL] Workshop on Domain Specific Visual Languages; "Results Poster". Held at OOPSLA 2001 (organizers: Tolvanen, Gray, Kelly and Lyytinen) http://www.isis.vanderbilt.edu/oopsla2k1/Presentations/ ResultsOOPSLA-DSVL-2001.ppt

[XSLT] W3C; "XSL Transformations version 1.0". URL: http://www.w3.org/TR/xslt.