Verifying Model Transformations by Structural Correspondence

Anantha Narayanan[1] **and Gabor Karsai**[1]

[1]Institute for Software Integrated Systems,
Vanderbilt University
Nashville, TN 37203 USA

Abstract: Model transformations play a significant role in model based software development, and the correctness of the transformation is crucial to the success of the development effort. We have previously shown how we can use bisimulation to verify the preservation of certain behavioral properties across a transformation. However, transformations are often used to construct structurally different models, and we might wish to ensure that there is some structural correspondence to the original model. It may be possible to verify such transformations without having to explicitly specify the dynamic semantics of the source and target languages. In this paper, we present a technique to verify such transformations, by first specifying certain structural correspondence rules between the source and target languages, and extending the transformation so that these rules can be easily evaluated on the instance models. This will allow us to conclude if the output model has the expected structure. The verification is performed at the instance level, meaning that each execution of the transformation is verified. We will also look at some examples using this technique.

Keywords: Verification, Model transformations

1 Introduction

Model transformations that translate a source model into an output model are often expressed in the form of rewriting rules, and can be classified according to a number of categories [MV05]. However, the correctness of a model transformation depends on several factors, such as whether the transformation terminates, whether the output model complies with the syntactical rules of the output language, and others. One question crucial to the correctness of a transformation is whether it achieved the intended result of mapping the semantics of the input model into that of the output model. For instance, a transformation from a Statechart model to a non-hierarchical FSM model can be said to be correct if the output model truly reproduces the behavior of the original Statechart model.

Models can also be seen as attributed and typed graph structures that conform to an abstract syntax. Model transformations take an input graph and produce a modified output graph. In a majority of these cases, the transformation matches certain graph structures in the input graph and creates certain structures in the output graph. In such cases, correctness may be defined as whether the expected graph structures were produced in the output corresponding to the relevant structures in the input graph. If we could specify the requirements of such correspondences and trace the correspondences easily over instance models, a simple model checking process at the

end of a transformation can be used to verify if those instances were correctly transformed. In this paper, we explore a technique to specify *structural correspondence* rules, which can be used to decide if the transformation resulted in an output model with the expected structure. This will be specified along with the transformation, and evaluated for each execution of the transformation, to check whether the output model of that execution satisfies the correspondence rules.

2 Background

2.1 GReAT

GReAT [AKL03] is a language framework for specifying and executing model transformations using graph transformation. It is a meta-model based transformation tool implemented within the framework of GME [LBM+01]. One of the key features of GReAT is the ability to define cross-language elements by composing the source and target meta-models, and introducing new vertex and edge types that can be temporarily used during the transformation. Such cross-meta-model associations are called *cross-links*. Note that a similar idea is present in Triple Graph Grammars [Sch95]. This ability of GReAT allows us to track relationships between elements of the source and target models during the course of the transformation, as the target model is being constructed from the source model. This feature plays a crucial role in our technique to provide assurances about the correctness of a model transformation.

2.2 Instance Based Verification of Model Transformations

Verifying the correctness of model transformations in general is as difficult as verifying compilers for high-level languages. But for practical purposes, a transformation may be said to have 'executed correctly' if a certain instance of its execution produced an output model that preserved certain properties of interest. We call that instance 'certified correct'. This idea is similar to the work of Denney and Fischer in [DF05], where a program generator is extended to produce logical annotations necessary for formal verification of certain safety properties. An automated theorem prover uses these annotations to find proofs for the safety properties for the generated code. Note that this does not prove the safety of the code generator, but only of a particular instance of generated code.

In our previous effort [NK06], we have shown that it is both practical and prudent to verify the correctness of every execution of a model transformation, as opposed to finding a correctness proof for the transformation specification. This makes the verification tractable, and can also find errors introduced during the implementation of a transformation that may have been specified correctly.

This technique was applied to the specific case of preservation of reachability related properties across a model transformation. Reachability is a fairly well-understood property, and can be verified easily for a labeled transition system (LTS), for instance by model checking [Hol97]. If two labeled transition systems are *bisimilar*, then they will have the same reachability behavior. In our approach, we treated the source and target models as labeled transition systems, and verified the transformation by checking if the source and target instances were bisimilar.

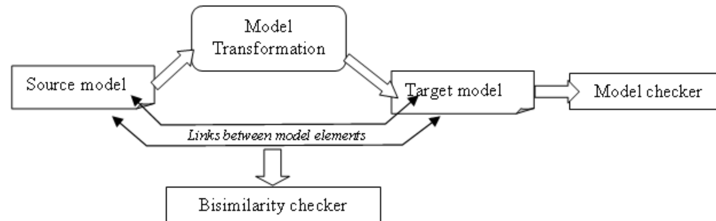


Figure 1: Architecture for Verifying Reachability Preservation in a Transformation

Given an LTS (S, Λ, $\rightarrow$), a relation $R \subseteq S \times S$ is a *bisimulation* [San04] if:

$$(p,q) \in R \text{ and } p \overset{\alpha}{\rightarrow} p' \text{ implies that there exists a } q' \in \text{S}, \\ \text{such that } q \overset{\alpha}{\rightarrow} q' \text{ and } (p',q') \in R,$$

and conversely,

$$q \overset{\alpha}{\rightarrow} q' \text{ implies that there exists a } p' \in \text{S}, \\ \text{such that } p \overset{\alpha}{\rightarrow} p' \text{ and } (p',q') \in R.$$

We used cross-links to relate source and target elements during the construction of the output model. These relations were then passed to a bisimilarity checker, which determined whether the source and target instances were bisimilar. If the instances were determined to be bisimilar, we could conclude that the execution of the transformation was correct. Figure 1 shows an overview of the architecture for this approach.

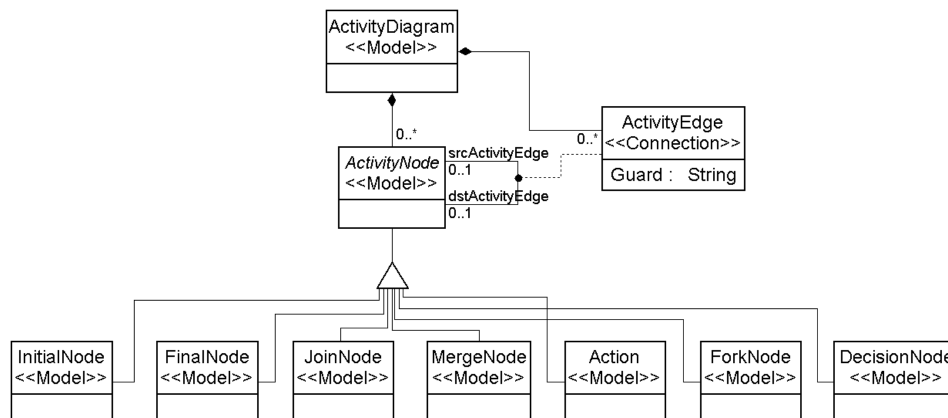


Figure 2: Meta-model for UML Activity Diagrams

2.3 UML to CSP Transformation

The UML to CSP transformation was presented as a case study at AGTIVE ’07 [BEH07], to compare the various graph transformation tools available today. We provide an overview of this

case study here from a GReAT point of view, and we will use this as an example to explain our technique for verifying model transformations.

The objective of this transformation is to take a UML Activity Diagram [OMG06] and generate a Communicating Sequential Process [Hoa78] model with the equivalent behavior. The Activity Diagram consists of various types of *Activity Nodes*, which can be connected by directed *Activity Edges*. Figure 2 shows the GME meta-model for UML Activity Diagrams. A CSP *Process* is defined by a *Process Assignment*, which assigns a *Process Expression* to a *Process Id*. *Process Expressions* can be a simple *Process*, a *Prefix* operator or a *BinaryOperator*. Figure 3 shows the GME meta-model for CSP, highlighting the relevant parts for our example.

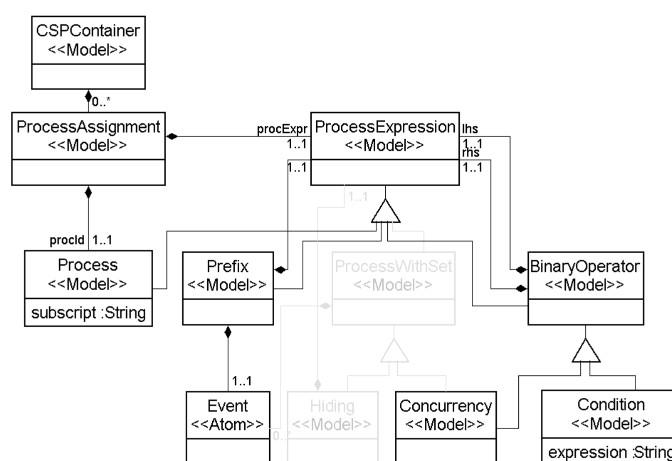


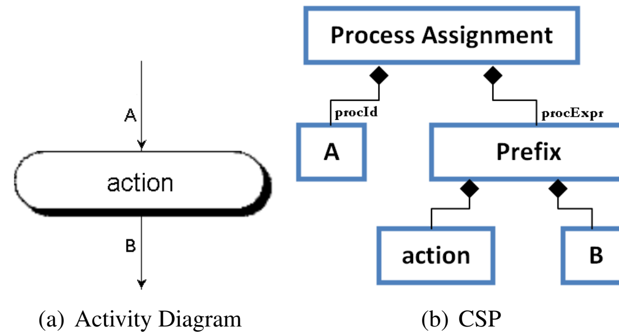
Figure 3: Meta-model for CSP

The UML to CSP mapping assigns a *Process* for each *Activity Edge*. For each type of *Activity Node*, a *Process Assignment* is created, which assigns the Process corresponding to the incoming Activity Edge to a *Process Expression* depending on the type of the Activity Node. Figure 4 shows one such mapping, for the type *Action Node*. This assigns the incoming Process to a *Prefix* expression. The resulting CSP expression can be written as $A = action \longrightarrow B$, which is shown in Figure 4 as a model instance compliant with the CSP meta-model.

3 Structural Correspondence

As in the UML to CSP case, model transformations can be used to generate a target model of a certain structure (CSP) from a source model of a different structure (UML). Specific structural configurations in the source model (such as an *Action Node* in the UML model) produce specific structural configurations in the target model (such as a *Prefix* in the CSP model). The rules to accomplish the structural transformations may be simple or complicated. However, it is fairly straightforward to compare and verify that the correct structural transformation was made, if we already know which parts of the source structure map to which parts of the target structure.

In our technique to verify a transformation by structural correspondence, we will first define



(a) Activity Diagram (b) CSP

Figure 4: CSP Process Assignment for Action Node

a set of structural correspondence rules specific to a certain transformation. We will then use cross-links to trace source elements with the corresponding target elements, and finally use these cross-links to check whether the structural correspondence rules hold.

In essence, we expect that the correspondence conditions are independently specified for a model transformation, and an independent tool checks if these conditions are satisfied by the instance models, after the model transformation has been executed. In other words, the correspondence conditions depend purely on the source and target model structures and not on the rewriting rules necessary to effect the transformation. Since the correspondence conditions are specified in terms of simple queries on the model around previously chosen context nodes, we expect that they will be easier to specify, and thus more reliable than the transformation itself. We also assume that the model transformation builds up a structure for bookkeeping the mapping between the source and target models.

3.1 Structural Correspondence Rules for UML to CSP Transformation

A structural correspondence rule is similar to a precondition-postcondition style axiom. We will construct them in such a way that they can be evaluated easily on model instances. We will use the UML to CSP example to illustrate structural correspondence. Consider the case for the *Action Node*, as shown in Figure 4. The *Action Node* has one incoming edge and one outgoing edge. It is transformed into a *Process Assignment* in the CSP. The CSP structure for this instance consists of a *Process Id* and a *Prefix*, with an *Event* and a target *Process*. This is the structural transformation for each occurrence of an Action Node in the Activity Diagram.

We can say that for each Action Node in the Activity Diagram, there is a *corresponding* Process Assignment in the CSP, with a Prefix Expression. When our transformation creates this corresponding Process Assignment, we can use a cross-link to track this correspondence. The structural correspondence is still not complete, as we have to ensure that the Process Id and the Prefix Expression are created correctly. We use a kind of *path expression* to specify the correctness of corresponding parts of the two structures, and the correspondence is expressed in the form $SourceElement = OutputElement$. Let us denote the Action Node by AN, and the Process Assignment by PA. Then the necessary correspondence rules can be written using path expressions as shown in Table 1.

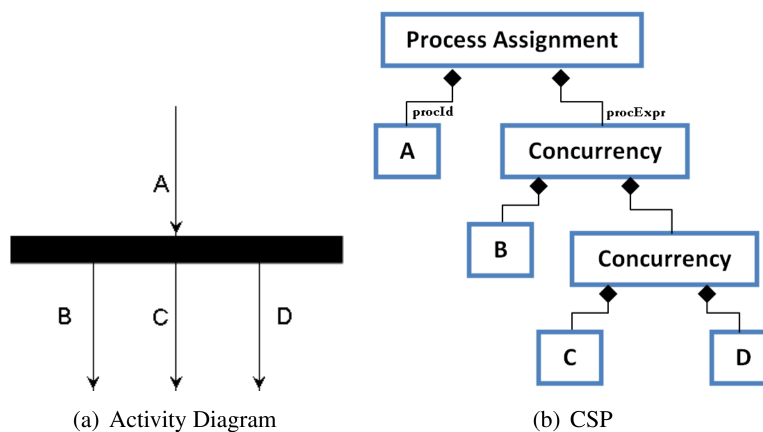
Rule	Path expression
The *Action Node* corresponds to a Process Assignment with a Prefix	$PA.procExpr.type = Prefix$
The incoming edge in the UML corresponds to the Process Id	$AN.inEdge.name = PA.procId.name$
The outgoing edge corresponds to the target Process	$AN.outEdge.name = PA.procExpr.Process.name$
The *action* of the Action Node corresponds to the Event	$AN.action = PA.procExpr.event$

Table 1: Structural Correspondence Rules for Action Node

These rules together specify the complete structural correspondence for a section of the Activity Diagram and a section of its equivalent CSP model. The different types of Activity Nodes result in different structures in the CSP model, some of which are more complex than the fairly straightforward case for the Action Node. Next, we look at the structural mapping for some of the other nodes.

A *Fork Node* in the Activity Diagram is transformed into a Process Assignment with a *Concurrency* Expression. Figure 5 shows a Fork Node with an incoming edge A and three outgoing edges B, C and D. This is represented by the CSP expression $A = B \parallel (C \parallel D)$, where $\parallel$ represents concurrency (the actual ordering of B, C and D is immaterial). The structural representation of this expression as an instance of the CSP meta-model is shown in Figure 5. The Fork Node is transformed into a CSP Process Assignment that consists of a Process Id corresponding to the incoming Activity Edge of the Fork Node, and a Process Expression of type *Concurrency*. The Concurrency Expression consists of Processes and other Concurrency nodes, depending on the number of outgoing Activity Edges.

If we denote the Fork Node by FN and the Process Assignment by PA, the structural corre-

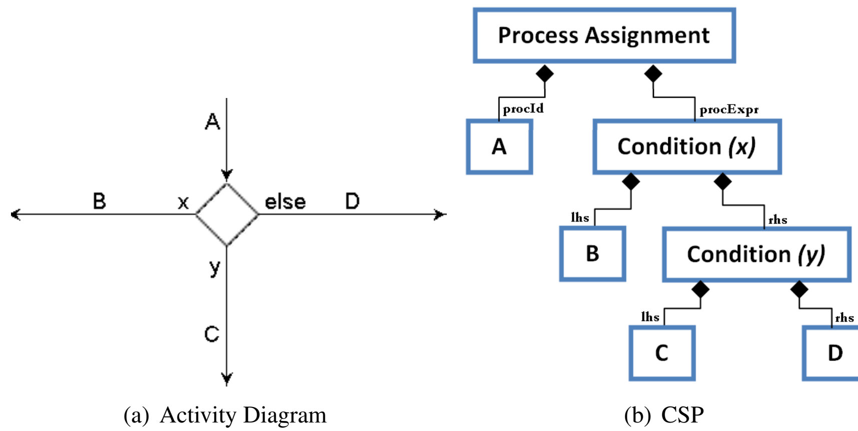


(a) Activity Diagram (b) CSP

Figure 5: CSP Process Assignment for Fork Node

Rule	Path expression
The *Fork Node* corresponds to a Process Assignment with a Concurrency	$PA.procExpr.type = Concurrency$
The incoming edge in the UML corresponds to the Process Id	$FN.inEdge.name = PA.procId.name$
For each outgoing edge, there is a corresponding Process in the Process Expression	$\forall o \in FN.outEdge$ $\exists p \in PA.procExpr..Process : o.name = p.name$

Table 2: Structural Correspondence Rules for Fork Node



(a) Activity Diagram

(b) CSP

Figure 6: CSP Process Assignment for Decision Node

spondence rules can be described using path expressions as shown in Table 2. We will use the double-dot '..' to denote the *descendant* operator (similar to '//' in XPath queries), to specify ancestor-descendant relationships.

By evaluating these rules on the Activity Diagram and the CSP models, we can determine whether the structural correspondence was satisfied for Fork Nodes. Another type of node in the Activity Diagram is the *Decision Node*. The transformation mapping for the Decision Node is a slight variation of the Fork Node. Figure 6 shows a Decision Node with an incoming edge A, and three outgoing edges B, C and D, with guards x, y and *else* respectively (in this case study, we will assume that the Decision Node always has exactly one 'else' edge). This is represented by the CSP expression $A = B \nless x \ngtr (C \nless y \ngtr D)$, where the operator $C \nless y \ngtr D$ is the *condition* operator with the meaning that if y is *true*, then the process behaves like C, else like D.

The Decision Node is transformed to the CSP model shown in Figure 6 as a model instance of the CSP meta-model. The Decision Node is transformed into a Process Assignment that consists of a Process Id corresponding to the incoming Activity Edge of the Decision Node, and a Process Expression of type *Condition*. The Condition's *expression* attribute is set to the *guard* of the corresponding Activity Edge, and a Process corresponding to the Activity Edge is created as it's LHS. For the final 'else' edge, a Process is created in the last Condition as it's RHS. The

Rule	Path expression
The *Decision Node* corresponds to a Process Assignment with a Condition	$PA.procExpr.type = Condition$
The incoming edge in the UML corresponds to the Process Id	$DN.inEdge.name = PA.procId.name$
For each outgoing edge, there is a corresponding Condition in the Process Expression and a corresponding Process in the Condition's LHS	$\forall o \in DN.outEdge \wedge o.guard \neq else$ $\exists c \in PA.procExpr..Condition :$ $c.expression = o.guard \wedge c.lhs.name = o.name$
For the outgoing 'else' edge, there is a Condition in the Process Expression with a corresponding Process as it's RHS	$\forall o \in DN.outEdge \wedge o.guard = else$ $\exists c \in PA.procExpr..Condition :$ $c.rhs.name = o.name$

Table 3: Structural Correspondence Rules for Decision Node

structural correspondence rules for this mapping are shown in Table 3.

3.2 Specifying Structural Correspondence Rules in GReAT

Specifying the structural correspondence for a transformation consists of two parts:

1. Identifying the significant source and target elements of the transformation
2. Specifying the structural correspondence rules for each element using path expressions

The first step is accomplished in GReAT by using cross-links between the source and target elements. A composite meta-model is added to the transformation, by associating selected source and target elements using a temporary 'Structural Correspondence' class. There will be one such class for each pair of elements. This class will have a string attribute, which is set to the path expressions necessary for structural correspondence for that pair. Figure 7 shows a composite meta-model specifying the structural correspondence for Fork Nodes. The *Fork Node* class comes from the Activity Diagram meta-model, and the *Process Assignment* class comes from the CSP meta-model.

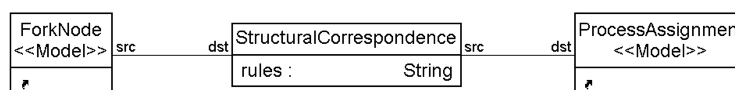


Figure 7: Composite Meta-model to Specify Structural Correspondence

Once the structural correspondence has been specified for all the relevant items, the transformation is enhanced to create the cross-link when creating the respective target elements. Figure 8 shows the GReAT rule in which the Process Assignment for a Fork Node is created, enhanced to

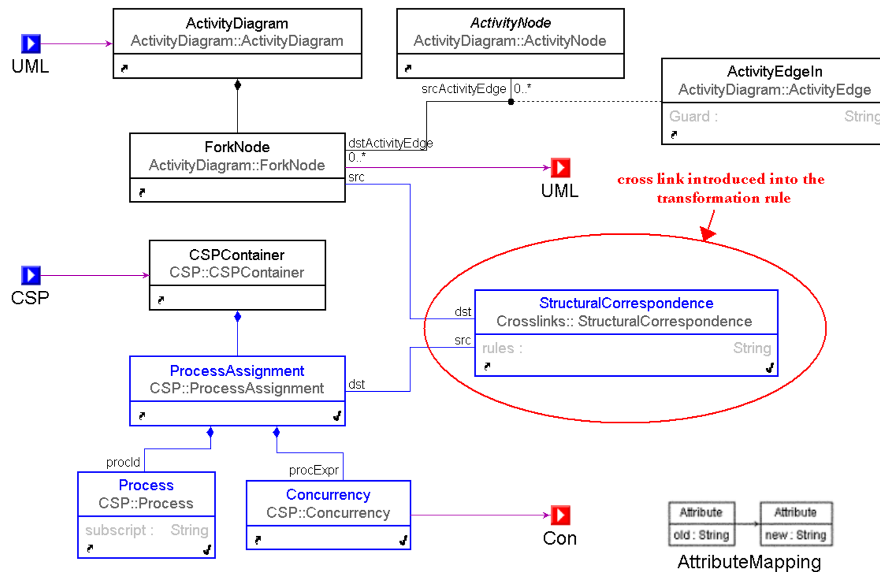


Figure 8: GReAT Rule with Cross-link for Structural Correspondence

create the cross-link for structural correspondence. Note that in incoming Activity Edge is represented as an *Association Class* named `ActivityEdgeIn`. The transformation for the Fork Node is actually accomplished in a sequence of several rules executed recursively, as shown in Figure 9. First all the Fork Nodes are collected, and a sequence of rules are executed for each node. These rules iterate through the out-edges of each Fork node, creating the Concurrency tree. Though several rules are involved in the transformation for Fork nodes, the cross-link needs to be added to one rule only.

It must be noted that it is necessary to specify the structural correspondence rules only once in the composite meta-model. The cross-link must however be added to the transformation rules, and in most cases will be required only once for each pair of source and target element.

3.3 Evaluating the Structural Correspondence Rules

Once the structural correspondence rules have been specified, and the cross-links added to the transformation, the correspondence rules are evaluated on the instance models after each execution of the transformation. These rules can be evaluated by performing a simple depth first search on the instance models, and checking if the correspondence rules are satisfied at each relevant stage.

This consists of two phases. The first phase is to generate the code that will traverse the instance models and evaluate the correspondence rules. Since the meta-models of both the source and target languages are available with the transformations, and the path expressions are written in a standard form that can be parsed automatically, the model traverser code can be automatically generated from the structural correspondence specification. This needs to be done only once each

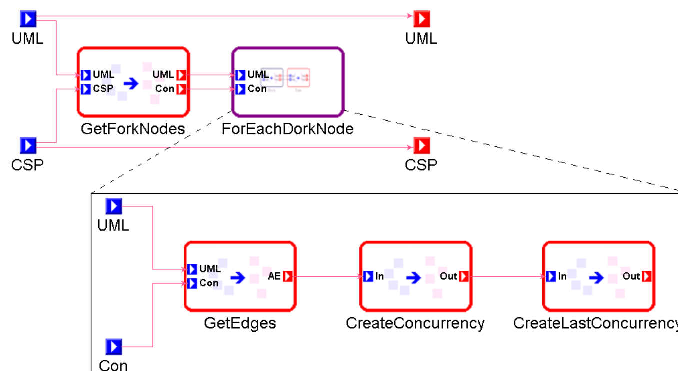


Figure 9: Sequence of GReAT Rules for Fork Node Transformation

time the structural correspondence specification changes. The second phase is to call the model traverser code at the end of each execution of the transformation, supplying to it the source and target model instances along with the cross-links.

In the case of the UML to CSP transformation, we traverse the input Activity Diagram model and evaluate the correspondence rules at each activity node. For each Activity Node, the cross-link is traversed to find the corresponding Process Assignment. If a correspondence rule has been defined for an Activity Node, and no corresponding Process Assignment is found, then this signals an error in the transformation. After locating the corresponding Process Assignment, the path expressions are evaluated. If any of the rules are not satisfied, the error is reported. If all the rules are satisfied for all the nodes, then we can conclude that the transformation has executed correctly.

The instance model is traversed in a depth-first manner. The corresponding elements are located using the cross-links, which will take constant time. The path expressions are evaluated on the instances, which will take polynomial time in most cases. Thus, the overall verification process does not incur a significant performance overhead in most cases.

3.4 Remarks

The structural correspondence based verification described here can provide important assurances about the correctness of transformations, while being practically applicable in most common transformations.

The use of path expressions to specify correspondence rules makes it easy to specify correctness. The path expressions use a simple query language that can be easily evaluated on the instance models. Our future research concentrates on the requirements of such a query language. Most complex transformations may involve multiple rules executing recursively to transform a particular part of a model. However, it may be possible to specify the correspondence for that part of the model using a set of simple path expressions. Such a specification would be simpler and easier to understand than the complex transformation rules.

The structural correspondence is also specified orthogonal to the transformation specification.

Thus, these rules may be written by someone other than the original transformation writer, or even supplied along with the requirements document.

4 Related Work

[KÖ4], [KHE03] present ideas on validating model transformations. In [KHE03], the authors present a concept of rule-based model transformations with control conditions, and provide a set of criteria to ensure termination and confluence. In [KÖ4], Küster focuses on the syntactic correctness of rule-based model transformations. This validates whether the source and target parts of the transformation rule are syntactically correct with respect to the abstract syntax of the source and target languages. These approaches are concerned with the functional behavior and syntactic correctness of the model transformation. [de] also discusses validation of transformations on these lines, but also introduces ideas of syntactic consistency and behavior preservation. Our technique addresses semantic correctness of model transformations, addressing errors introduced due to loss or mis-representation of information during a transformation.

In [BH07], Bisztray and Heckel present a rule-level verification approach to verify the semantic properties of business process transformations. CSP is used to capture the behavior of the processes before and after the transformation. The goal is to ensure that every application of a transformation rule has a known semantic effect. We use path expressions to capture the relation between structures before and after a transformation. These path expressions are generic (they do not make any assumptions about the underlying semantics of the models involved), and can be applied to a wide variety of transformations.

Ehrig et. al. [EEE+07] study bidirectional transformations as a technique for preserving information across model transformations. They use triple graph grammars to define bi-directional model transformations, which can be inverted without specifying a new transformation. Our approach offers a more relaxed framework, which will allow some loss of information (such as by abstraction), and concentrates on the crucial properties of interest. We also feel that our approach is better suited for transformations involving multiple models and attribute manipulations.

In other related work, [VP03] presents a model level technique to verify transformations by model checking a selected semantic property on the source model, and transforming the property and validating it in the target domain. The validation requires human expertise. After transforming the property, the target model is model checked. In our approach, since the properties are specified using cross links that span over both the source and target languages, we do not need to transform them. [LLMC06] discuss an approach to validate model transformations by applying OCL constraints to preserve and guarantee certain model properties. [GGL+06] is a language level verification approach which addresses the problem of verifying semantic equivalence between a model and a resulting programming language code.

4.1 MOF QVT Relations Language

The MOF 2.0 Query / View / Transformation specification [OMG05] addresses technology pertaining to manipulation of MOF models. A *relations language* is prescribed for specifying relations that must hold between MOF models, which can be used to effect model transformations.

Relations may be specified over two or more domains, with a pair of *when* and *where* predicates. The *when* predicate specifies the conditions under which a relation must hold, and the *where* predicate specifies the condition that all the participating model elements must satisfy. Additionally, relations can be marked as *checkonly* or *enforced*. If it is marked *checkonly*, the relation is only checked to see if there exists a valid match that satisfies the relationship. If it is marked *enforced*, the target model is modified to satisfy the relationship whenever the check fails.

Our approach can be likened to the *checkonly* mode of operation described above. However, in our case, the corresponding instances in the models are already matched using cross links, and the correspondence conditions are evaluated using their context. The cross links help us to avoid searching the instances for valid matches. Specifying the correspondence conditions using context nodes simplifies the model checking necessary to evaluate the conditions, thus simplifying the verification process. Since we verify the correspondence conditions for each instance generated by the transformation, these features play an important role.

4.2 Triple Graph Grammars

Triple Graph Grammars [Sch95] are used to describe model transformations as the evolution of graphs by applying graph rules. The evolving graph complies with a graph schema that consists of three parts. One graph schema represents the source meta model, and one represents the target meta-model. The third schema is used to track correspondences between the source and target meta models. Transformations are specified declaratively using triple graph grammar rules, from which operational rules are derived to effect model transformations.

The schema to track correspondences between the source and target graphs provides a framework to implement a feature similar to cross links in GReAT. If the correspondence rules can be encoded into this schema, and the correspondence links persisted in the instance models, our verification approach can be implemented in this scenario.

5 Conclusions and Future Work

In this paper, we have shown how we can provide an assurance about the correctness of a transformation by using structural correspondence. The main errors that are addressed by this type of verification is the loss or misrepresentation of information during a model transformation.

We continue to hold to the idea that it is often more practical and useful to verify transformations on an instance basis. The verification framework must be added to the transformation only once, and is invoked for each execution of the transformation. The verification process does not add a significant overhead to the transformation, but provides valuable results about the correctness of each execution.

The path expressions must use a simple and powerful query language to formulate queries on the instance models. While existing languages such as OCL may be suitable for simple queries, we may need additional features, such as querying children to an arbitrary depth. Our future research concentrates on the requirements of such a language.

While the path expressions can be parsed automatically and evaluated on the instances, the cross-link for the relevant elements must be manually inserted into the appropriate transformation

rules. However, in most cases, it may be possible to infer where the cross-links must be placed. If the cross-links could be inserted into the rules automatically, the transformation can remain a black box. The main concern with this is that the cross-links are crucial to evaluating the correspondence rules correctly and also to keep the complexity down.

We have seen simple string comparisons added to the path expressions in this paper. Some transformations may require more complex attribute comparisons, or structure to attribute comparisons such as counting. We wish to explore such situations in further detail in future cases, to come up with a comprehensive language for specifying the path expressions.

Bibliography

[AKL03] A. Agrawal, G. Karsai, A. Ledeczi. An end-to-end domain-driven software development framework. In *OOPSLA '03: 18th annual ACM SIGPLAN conference on OOP, systems, languages, and applications*. Pp. 8–15. ACM Press, New York, NY, USA, 2003.
doi:http://doi.acm.org/10.1145/949344.949347

[BEH07] D. Bisztray, K. Ehrig, R. Heckel. Case Study: UML to CSP Transformation. In *Applications of Graph Transformation with Industrial Relevance (AGTIVE)*. 2007.

[BH07] D. Bisztray, R. Heckel. Rule-Level Verification of Business Process Transformations using CSP. In *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'07)*. 2007.

[de] de Lara, J. and Taentzer, G. Automated Model Transformation and its Validation with AToM3 and AGG. In *Lecture Notes in Artificial Intelligence, 2980*. Pp. 182–198. Springer.

[DF05] E. Denney, B. Fischer. Certifiable Program Generation. In Glück and Lowry (eds.), *GPCE*. Lecture Notes in Computer Science 3676, pp. 17–28. Springer, 2005.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In *Fundamental Approaches to Software Engineering*. Pp. 72–86. 2007.
doi:http://dx.doi.org/10.1007/978-3-540-71289-3_7

[GGL+06] H. Giese, S. Glesner, J. Leitner, W. Schfer, R. Wagner. Towards Verified Model Transformations. October 2006.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM* 21(8):666–677, 1978.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.* 23(5):279–295, 1997.
doi:http://dx.doi.org/10.1109/32.588521

[KÖ4] J. M. Küster. Systematic Validation of Model Transformations. In *Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004)*. October 2004.

[KHE03] J. M. Küster, R. Heckel, G. Engels. Defining and validating transformations of UML models. In *HCC '03: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*. Pp. 145–152. IEEE Computer Society, Washington, DC, USA, 2003.

[LBM+01] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai. Composing Domain-Specific Design Environments. *Computer* 34(11):44–51, 2001. doi:http://dx.doi.org/10.1109/2.963443

[LLMC06] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf. Model-Based Development with Strictly Controlled Model Transformation. In *The 2nd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006*. Pp. 39–48. Paphos, Cyprus, May 2006.

[MV05] T. Mens, P. Van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation*. 2005.

[NK06] A. Narayanan, G. Karsai. Towards Verifying Model Transformations. In Bruni and Varro (eds.), *Graph Transformation and Visual Modeling Techniques GT-VMT 2006*. Electronic Notes in Theoretical Computer Science, pp. 185–194. 2006.

[OMG05] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[OMG06] OMG. Unifed Modeling Language, version 2.1.1. 2006. "http://www.omg.org/technology/documents/formal/uml.htm"

[San04] D. Sangiorgi. Bisimulation: From The Origins to Today. In *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*. Pp. 298–302. IEEE Computer Society, Washington, DC, USA, 2004. doi:http://dx.doi.org/10.1109/LICS.2004.13

[Sch95] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*. Pp. 151–163. Springer-Verlag, London, UK, 1995.

[VP03] D. Varró, A. Pataricza. Automated Formal Verification of Model Transformations. In Jürjens et al. (eds.), *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*. Technical Report TUM-I0323, pp. 63–78. Technische Universität München, September 2003.