Specifying the Correctness Properties of Model Transformations

Anantha Narayanan
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
anantha.narayanan@vanderbilt.edu

Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
gabor.karsai@vanderbilt.edu

ABSTRACT

The correctness of a model transformation is central to the success of a model-driven software development process. A transformation can be said to have executed correctly if it resulted in the desired output model, but this requires a specification of what constitutes a desirably correct output. If we have this specification, and a framework to verify that it holds on a specific execution of the transformation, then that execution instance may be "certified correct". In this paper, we explore a technique to specify such a correctness, using a language framework that can easily be incorporated into a variety of domains. We will also see how these correctness criteria can be verified on instance models.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

Keywords

Model Transformations, Verification

1. INTRODUCTION

Model based software development has advanced to a level of maturity where most artifacts in the design and development stages could be produced by automated model transformations. The success of such a development effort hinges on the correctness of these automated model transformations. The term *correctness*, in this context, involves several facets such as termination of the transformation algorithm, confluence of the transformation rules and conformance to a language syntax or meta-model. One important criterion for correctness is whether the transformation achieved its objective, whether it resulted in the desired output model. While the former properties can be stated implicitly within the framework of the transformation language, the notion of correctness in the latter case involves a more detailed knowledge of the source and target domains and the transformation objective from a domain-specific point of view. We present a technique to specify such a correctness, using a language framework that can easily be incorporated into a variety of domains.

Models can be seen as typed, attributed graphs, and model transformations as manipulations on such graph structures. In other words, a source model of a certain graph structure is transformed into a target model of a different structure. In most transformation cases where the correctness problem is significant, there is a correlation or *correspondence* between parts of the input model and parts of the output model. If we can specify these correlations in terms of the abstract syntax of the source and target languages, and have a framework to verify whether the correlations hold, then we can verify whether the desired output model was created. We call these correlations *structural correspondence*. Our thesis for the approach is as follows: if a transformation has resulted in the desired output models, there will be a verifiable structural correspondence between the source and target model instances that is decidable. This idea was introduced in [12] with a simple case study. In this paper, we will concentrate on the requirements of a query language for specifying correctness in this form, and look at more detailed examples that illustrate these requirements.

2. BACKGROUND

2.1 Instance based verification

In our previous work on verifying model transformations [11], we used the underlying behavior models of the source and target languages, and framed the verification problem in terms of finding a bisimulation between the source and target models. One key feature of the approach was that we focused on verifying each execution of the transformation, as opposed to finding a correctness proof for the transformation specification. We believe that this approach is more simple and pragmatic. Figure 1 shows the framework we used.

We extended the transformation framework to trace relations between elements of the source and target instances. At the end of the transformation's execution, we passed these links to a bisimilarity checker, which checked if there was a bisimulation between the source and target models. This helped us to conclude whether a behavioral property was preserved by that execution of the transformation. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GraMoT'08, May 12, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-033-3/08/05 ...$5.00.

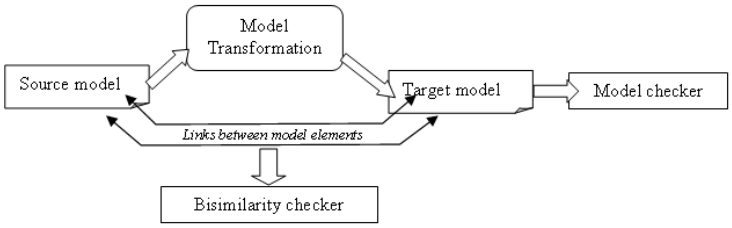


Figure 1: Instance Level Verification of Transformations

instance model is then said to be 'certified correct'. Note that while the verification is performed for each execution of the transformation, the transformation program had to be augmented with the additions for the verification framework only once.

While this example is concerned with behavioral properties, we will consider verification from a structural viewpoint in the rest of this paper. However, we will continue to focus on verifying each execution of a transformation instead of finding general correctness proofs.

2.2 GReAT

GReAT [1] is a language and transformation framework for specifying and executing meta-model based model transformations using graph transformation rules. GReAT is implemented with the framework of GME [9]. The source and target meta-models of the transformation are specified using UML class diagrams, and transformation rules are constructed using these classes. GReAT also allows transformation designers to create new vertex and edge types in addition to those defined in the source and target meta-models. The transformation can thus be written over a composite meta-model that consists of the source and target languages, and some cross-language elements. This allows us to track associations between elements of the source and target instances during the course of a transformation. Such associations are called *cross links*.

2.3 Class to RDBMS Transformation

The "Class to RDBMS" transformation example [2] was presented as a challenge problem in the 2005 Model Transformations in Practice Workshop. We will use this example to explain some of the ideas in this paper. A short description of the problem is presented here.

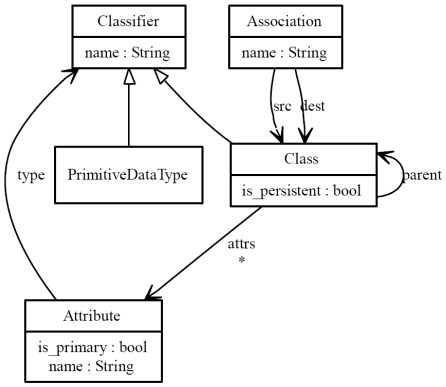


Figure 2: Class meta-model [2]

Figure 2 shows the meta model for Classes and Associations. Classes can contain one or more Attributes, with the additional constraint that there is at least one *primary* attribute. Figure 3 shows the meta model for RDBMS. An RDBMS model contains one or more Tables, each with one or more Columns. One or more of these Columns forms the *primary key* of the table. A table may contain zero or more *foreign keys*.

The goal of the transformation is to create an RDBMS representation from a class diagram, based on a number of rules. A *Table* is created for each top level *Class* in the source model which has its *is_persistent* attribute set to *true*. The *Attributes* and *Associations* of the class are transformed into *Columns* of the corresponding table, and the primary and foreign keys are also set appropriately. The complete details of the transformation requirements can be found in [2].

We will not attempt to provide a solution for this transformation here, we refer the reader to [5] for some solutions to the transformation. We will simply use it below to illustrate how a verification framework can be built around such a transformation.

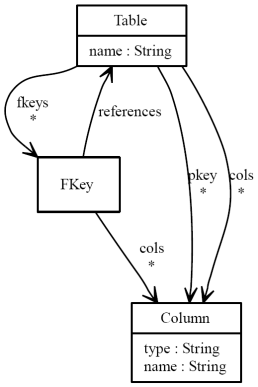


Figure 3: RDBMS meta-model [2]

3. SPECIFYING CORRECTNESS BY CORRESPONDENCE

We wish to ensure that a model transformation executed correctly, by verifying whether it produced the desired output model. To accomplish this, we need a specification of what constitutes a desired model. This specification must not be confused with the specification of the transformation itself. In most cases, it is sufficient that the output model satisfies a small number of constraints based on the source model elements and attributes, to be accepted as correct.

However, the transformation that produces such an output model may have to deal with intricate issues, and thus be much more complex. Further, with a simplified language for specifying such conditions, and the use of cross links, we can construct a pragmatic framework to verify the correctness of the output model.

3.1 Structural Correspondence

In typical model transformations, we frequently wish to create a structure in the output model corresponding to some structure in the input model. The transformation can be accepted as correct, if a node in the source model and its corresponding node in the target model satisfy some *correspondence conditions*. In this case, a tractable solution to the verification problem can be provided under the following conditions: 1. A map is maintained to match the corresponding nodes in the source and target model instances; 2. Correspondence conditions are specified in terms of these previously identified and matched nodes. Our approach consists of the following steps.

3.1.1 Identifying Correspondence Structures

The first task is to identify a sufficient set of node types from the source language that must have a corresponding element in the target model. We call these *pivot nodes*. While all the node types being transformed can be considered here, it may be pragmatic to only choose a smaller set of significant nodes - which either undergo complex transformations or are critical to the correctness of the output model.

For instance, in the Class to RDBMS transformation example, *Class* nodes are transformed into *Table* nodes, and *Attribute* nodes are transformed into *Column* nodes. These two pairs can be identified as the pivot nodes for this problem. The verification problem is to ensure that the *Tables* and *Columns* are correctly created in the output model corresponding to the *Classes* and *Attributes* in the source model. Cross links are defined between such pairs, which will carry the correspondence specifications and will later assist in the verification of the correspondence conditions.

3.1.2 Specifying Correspondence Conditions

Once the pivot nodes have been identified, the correctness condition must be specified for each pair of pivot nodes. The correctness condition can use some form of query that can be performed on the instance models. This query could involve traversing the immediate hierarchy of the nodes, access the nodes' attributes and associations etc. This is explained in greater detail in Section 3.2.

Nodes of type *Class* in the source model correspond to nodes of type *Table* in the target model. The correspondence rule must ensure that for every such pair in the model being transformed, the *is_persistent* attribute of the *Class* node is set to *true*. The correspondence must further state that there exists a *Column* for each *Attribute* of the class and one for each *Association* where the class acts as the source, which are in turn related by correspondence conditions. This is specified using existential quantifiers that make use of the cross link relation.

3.1.3 Creating cross links

Cross links are used to construct a look-up table (i.e. a map) that matches the corresponding pivot nodes of any instance of a transformation execution. The cross links are crucial to have a tractable and reliable verification framework. The transformation must be extended by creating a new cross link between the pivot nodes in every transformation rule that creates the relevant target node. The transformation has to be extended this way only once.

The cross links are specified at the meta level, by drawing a link between the source node and its corresponding target node. To clarify the specification of the correspondence conditions, we will use a *StructuralCorrespondence* class that carries the correspondence rules for a specific pair of source and target nodes. We can then connect the source and corresponding target nodes through this intermediate class using cross links, as shown in Figure 4. This gives rise to a composite meta model that includes the source and target languages along with the cross links. Since the transformation rules are defined by referencing the meta types in the pattern matching rule, it may be possible to insert the cross links automatically into the relevant rules of the transformation. For instance, a cross link is created between *Class* in the source meta and it Table in the target meta. Similarly, cross links are created between *Attribute* and *Column*, and *Association* and *Column*.

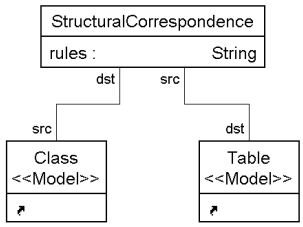


Figure 4: A cross link to specify structural correspondence

3.1.4 Checking the Correspondence Conditions

At the end of each transformation run the source and target instance models, along with the cross links, are passed to a model checker. The model checker uses these cross links to check if the verification conditions hold for all pairs of pivot nodes through out the model instances. This is explained in Section 3.3.

3.2 Design of a Query Language for Specifying Correspondence

It is important to note that the correspondence conditions are not intended to specify complete transformations using complex pattern matching, but simply list some conditions to help to satisfactorily conclude that the correct transformation was made. This simplifies the query language to a degree. Also, the verification conditions are checked on specific nodes as opposed to arbitrary subgraphs, and have the option of looking up matches using cross links. The instance model is traversed exhaustively, and a check is performed on each node that has a correspondence specification. Thus, there is a specific context in which each condition is evaluated. The query language is expected to have the following features.

3.2.1 Querying Attributes and Associations

It must be possible to reference the attributes and associations of the pivot nodes based on the abstract syntax of

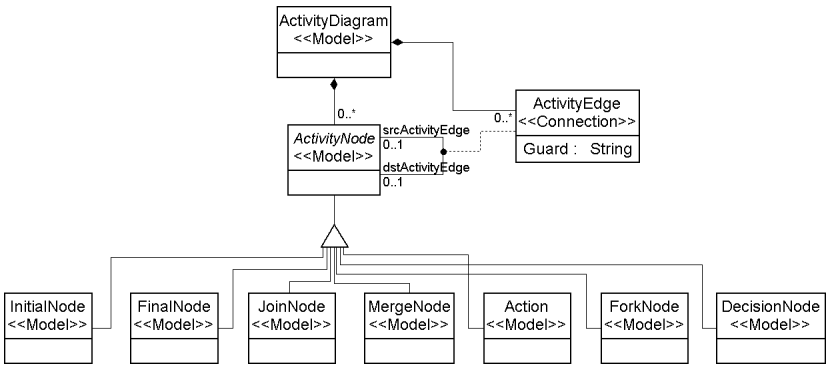


Figure 5: Meta-model for UML Activity Diagrams

the languages. A simple OCL style path query of the form *class.attribute* or *class.association* suffices here. Since they are executed per instance of the pivot node, it is always expected to terminate with a finite result.

3.2.2 Querying Up or Down a Containment Hierarchy

We may need to reference the pivot nodes' parent or child nodes to specify certain correspondence conditions. Most query languages (such as OCL) allow querying child nodes using the child role names. A similar notation for finding the pivot nodes' parents is required. Further, some structural correspondence conditions may require querying child nodes at an arbitrary depth. The query language can be extended to use a double dot '..' notation to query all contained child nodes at any level of depth in the hierarchy. Since model hierarchies cannot contain cycles, this query will always terminate with a finite result.

3.2.3 Using Quantifiers with Queries

Finally, we may need to add quantifiers to the query string to frame the correspondence conditions. For instance, we may want to use statements like 'a Column is created *for each* Attribute of a Class'. The standard quantifiers such as $\exists$ (there exists at least one), $\forall$ (for all) and $\exists!$ (there exists exactly one) can be used to specify most of the correspondence conditions. We conjecture that it is useful to also have specialized quantifiers that take advantage of correspondence information stored in the cross links, by imposing some restrictions on the standard quantifiers. For instance, $\exists_C$ can be used to represent 'there exists, attached via a cross link'. Such a statement is useful in cases where an object in the source model must correspond to a number of objects in the target model. This reduces the search space and simplifies the verification.

3.3 Checking the Verification Conditions

At the end of each execution of the transformation, the verification conditions are checked on the instance model. This is performed by a model checker that performs an exhaustive scan of the instance models, applying the verification conditions on all the relevant nodes. This model checker must be tailored to traversing instance models of the relevant domains and evaluating the relevant verification conditions. We propose the use of a generic model checker, that can be customized automatically using the meta-models of the source and target languages, and the verification conditions specified in a standardized form. The steps of the generic model checker are listed in Algorithm 3.1.

Algorithm 3.1: checkCorrespondence($instances$, $crosslinks, metamodels, correspondence-rules$)

```
for each instance node
  if type has correspondence rules  then
    find corresponding target nodes
    if failed return (false)
    for each rule
      Evaluate (rule)
      if failed return (false)
    end for
  end if
end for
return (true)
```

4. CASE STUDY: UML TO CSP TRANSFORMATION

In this section, we look at some application examples of verification by structural correspondence, using a more detailed case study. We will consider a transformation from UML Activity Diagrams [14] to a CSP (Communicating Sequential Processes) [6] specification. Figure 5 shows the meta model for UML Activity Diagrams. Activity Diagrams consist of Activity Nodes and Activity Edges. Figure 6 shows the meta model for CSP. A CSP model consists of a number of Process Assignments, which assign a Process Id to some Process Expression. The Process Expression can be of various types such as Prefix, Concurrency, Condition etc.

The objective of the UML to CSP transformation is to obtain a CSP specification from an activity diagram, by creating CSP process assignments that mimic the behavior of the activity diagram. A CSP Process represents an Activity Edge, and is assigned to a Process Expression based on the type of the Activity Node involved. The complete description of this transformation can be found in [3]. We will look at some specific portions of the transformation here.

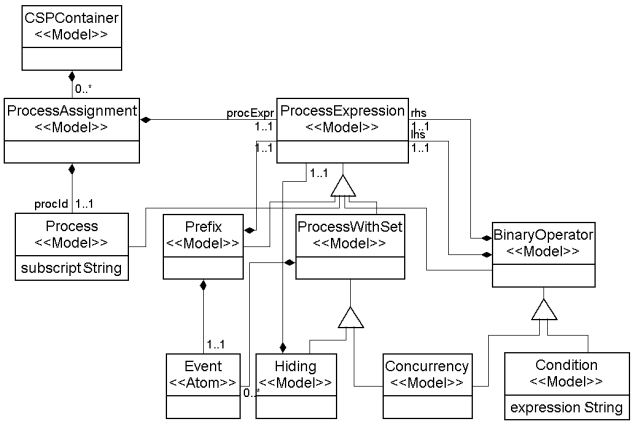
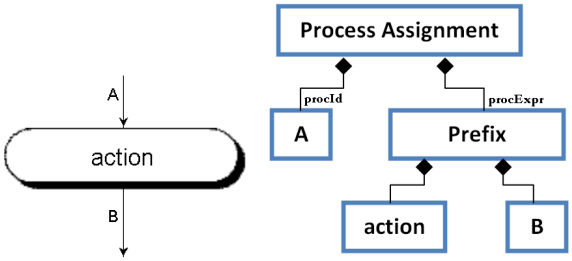


Figure 6: Meta-model for CSP

4.1 Action Nodes

Action Nodes in the activity diagram are represented by Process Assignments using Prefix expressions. Action Nodes have one incoming Activity Edge and one outgoing Activity Edge. The expected transformation for Action Nodes in the activity diagram is shown in Figure 7.



(a) Activity Diagram (b) CSP

Figure 7: CSP Process Assignment for Action Node

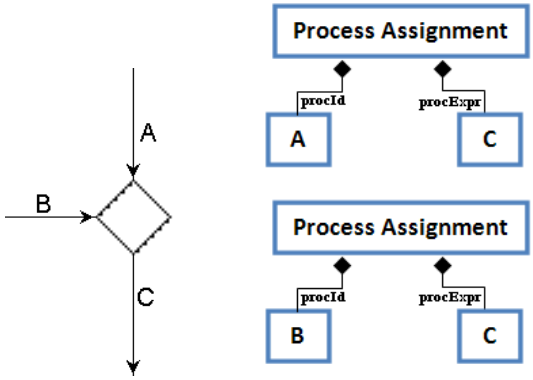
The condition for correspondence in this case is simply that each (UML) Action node has a corresponding (CSP) Process Assignment, with the Process Id set to the incoming edge of the Action Node, and the Process Expression is a Prefix expression that encodes the *action* of the Action node as its *event* and the outgoing edge of the Action node as its target process. To specify this correspondence, we create a cross link between the Action node and the Process Assignment, and encode the correspondence conditions for the cross link. This relates each instance of type Action Node in the activity diagram model to an instance of type Process Assignment in the CSP model, which is directly identified by the cross link between the instances. If we denote the (UML) Action node by AN and the corresponding (CSP) Process Assignment by PA, we will have the following conditions:

- $PA.procExpr.type = Prefix$
- $AN.inEdge.name = PA.procId.name$
- $AN.action = PA.procExpr.event$
- $AN.outEdge.name = PA.procExpr.process.name$

Note that this set of verification conditions is checked for each instance of an Action node in the model being transformed. At the end of transformation, the instance models are traversed, and for each AN found, the corresponding PA is located through the cross link (if no PA is found, that itself indicates an error in the transformation). With the AN and its corresponding PA, the verification conditions are checked. If all conditions are satisfied, we move on to the next object instance. If a condition is not satisfied, then an error has been found in the transformation.

4.2 Merge Nodes

Merge nodes merge multiple activity edges into a single activity edge. This is performed by simply assigning the process corresponding to each incoming edge to the process corresponding to the outgoing edge. Figure 8 shows the transformation for Merge nodes.



(a) Activity Diagram (b) CSP

Figure 8: CSP Process Assignment for Merge Node

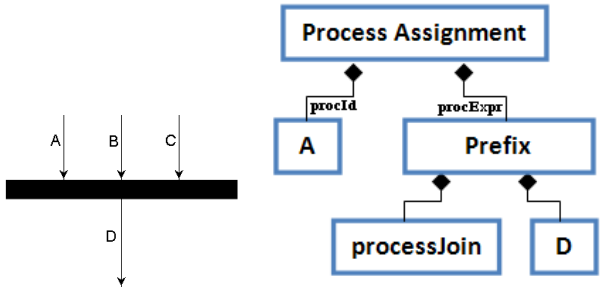
In this transformation, as many Process Assignments must be created as there are incoming edges for the merge node. To establish the correspondence, each of the Process Assignment nodes must be cross linked to the Merge node. When specifying the correspondence, however, we need to create only one cross link between the Merge Node type (in the Activity Diagram meta) and the Process Assignment type (in the CSP meta). The correspondence condition must ensure that the required number of Process Assignments were created, and that a Process corresponding to each of the incoming activity edges is assigned to a Process corresponding to the outgoing activity edge. To specify this correspondence, we make use of the specialized quantifier $\exists_C$. If a Merge Node is denoted by MN and a Process Assignment is denoted by PA:

- $\forall\ MN.inEdge\ \exists_C\ PA:$
  $PA.procId.name = MN.inEdge.name$
  $\wedge PA.procExpr.name = MN.outEdge.name$

The verification condition is checked for each occurrence of a Merge node in the instance model being transformed. Each MN will have a finite number of $inEdges$, and a finite number of cross-linked PAs, and the condition can be verified by a model checker by traversing the instance models. This example shows the advantage of having a specialized quantifier like $\exists_C$ that can use the information represented by cross links.

4.3 Join Nodes

Join Nodes can be thought of as the 'synchronized' version of Merge nodes. Join nodes also have multiple incoming activity edges and one outgoing activity edge, but represent the semantics that the associated processes must wait for each other before joining. This is represented in the CSP by using a Prefix expression and a special event *processJoin* that synchronizes the processes involved. The Join Node is transformed such that one Process (corresponding to some incoming edge) is chosen to go on with the continuation Process (corresponding to the outgoing activity edge), while the other Processes simply terminate in a $SKIP$. For the Join Node shown in Figure 9(a), the CSP process assignments would be: $A = processJoin \longrightarrow D; B = processJoin \longrightarrow SKIP; C = processJoin \longrightarrow SKIP$. Figure 9 shows the transformation for Join Nodes, with the expected CSP model structure for the the first Process Assignment. For the remaining Process Assignments, the target process D is replaced by $SKIP$.



(a) Activity Diagram (b) CSP

Figure 9: CSP Process Assignment for Join Node

Like in the case of Merge nodes, multiple Process Assignments must be created for each Join node, depending on the number of incoming edges incident on the Join node. But unlike the previous case, we have the additional condition that the target process corresponds to the outgoing edge for exactly one of the Process Assignments (it does not matter which), and is $SKIP$ for the rest of the Process Assignments. If a Join Node is denoted by JN and a Process Assignment by PA, we have the following conditions:

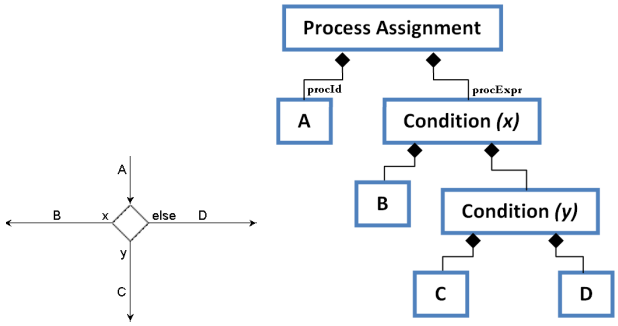
- $\forall\ JN.inEdge\ \ \exists_C\ \ PA$:
  $PA.procId.name = JN.inEdge.name$
  $\wedge\ PA.procExpr.type = Prefix$
  $\wedge\ PA.procExpr.event =$ '$processJoin$'
- $\forall\ JN.inEdge\ \ \exists!\ \ PA$:
  $PA.procExpr.process.name = JN.outEdge.name$
- $\forall\ JN.inEdge\ \ \exists!\ \ PA$:
  $PA.procExpr.process.name \neq SKIP$

These conditions are checked for each occurrence of a Join node in the instance model being transformed. The first condition checks whether a PA has been created corresponding to each incoming activity edge in the source model. The second condition checks if exactly one of these has the target expression set to a Process corresponding to the outgoing activity edge. The last condition ensures that there is only one such PA, and the rest of the processes terminate in a $SKIP$.

4.4 Decision Nodes

Decision nodes have one incoming edge and several outgoing edges, with a *guard* on each outgoing edge. We also assume that at least one of these guards is *else*. Decision nodes are represented using Condition expressions in the CSP. The CSP Condition is a binary expression, with an *lhs* and an *rhs* Process expression, and a condition expression that captures the guard for the condition. Due to the binary nature of the condition expression, decisions involving more than one condition must be represented using a binary tree structure. The Decision node shown in Figure 10(a) is represented by the CSP process assignment $A = B \not< x \not> (C \not< y \not> D)$. The statement $C \not< y \not> D$ represents a binary condition which states that if y is *true*, then the process behaves like C, else like D. Figure 10(b) shows the CSP structure for this transformation.

Each Decision node is transformed into a single Process Assignment, which can be identified using the cross link. To verify if the Decision node was transformed correctly, we must check that each outgoing activity edge is represented as the *lhs* of a corresponding Condition which has its *expression* set to the *guard* of the activity edge, and there exists a Condition whose *rhs* captures the *else* activity edge. Since we must traverse the binary tree to a child node at an arbitrary depth to verify these conditions, we will make use of the '..' notation. If a Decision Node is represented by DN and its corresponding Process Assignment is represented by PA, then we will have the following conditions:



(a) Activity Diagram (b) CSP

Figure 10: CSP Process Assignment for Decision Node

- $PA.procId.name = DN.inEdge.name$
- $PA.procExpr.type = Condition$
- $\forall o \in DN.outEdge \wedge o.guard \neq else$
  $\exists c \in PA.procExpr..Condition$:
  $c.expression = o.guard$
  $\wedge\ c.lhs.name = o.name$
- $\forall o \in DN.outEdge \wedge o.guard = else$
  $\exists c \in PA.procExpr..Condition$:
  $c.rhs.name = o.name$

These conditions are verified for each occurrence of a DN in the instance model being transformed. The corresponding PA is located using the cross link, and its children are navigated to check the verification conditions. Since the nodes are identified using the cross links, the complexity is not expected to be high.

Finally, the model checker is customized to evaluate these correspondence rules on the instance models. The generic model checker listed in Algorithm 3.1 is extended for this domain and these rules. The listing in Algorithm 4.1 shows an overview of the customized model checker steps.

We have seen a number of cases where the correctness of the transformation has been specified using a structural correspondence. The examples we have seen are only representative of the kind of conditions that can satisfactorily determine correspondence. The actual number and depth of detail of these conditions depends on the domain and the application. This brings us to the question of completeness of the correspondence specification. Further research is needed to study the nature of these correspondence conditions in complex scenarios. Completely specifying semantic properties is always a difficult problem due to the variabilities of different domains. We believe that our approach of using context nodes and correspondences simplifies the problem to an extent, and could prove to be sufficient to completely specify correspondences in most cases.

Algorithm 4.1: checkCorrespondence($instances$, $crosslinks$, $metamodels$, $correspondence\_rules$)

```
for each activity node N
  if type is Action then
    PA := follow_cross_link(N)
    if failed, return (false)
    /*Evaluate Rules*/
    if not (PA.procExpr.type = Prefix)
      return (false)
    if not (PA.procId.name = N.inEdge.name)
      return (false)
    if not (PA.procExpr.event = N.action)
      return (false)
    if not (PA.procExpr.process.name =
      N.outEdge.name) return (false)
    continue
  else if type is Merge then
    ...
  end if
end for
return (true)
```

Note that in each of these cases, the rules necessary to accomplish the transformation may be quite complex, requiring several transformation steps and possibly requiring recursion. However, the correspondence conditions are relatively simple to specify and check using a simple brute force model checking approach. Moreover, the same verification conditions are independent of the algorithm chosen for the transformation, and depend only on the source and target structures. This makes it possible to plug them into existing transformation solutions to verify their correctness.

5. RELATED WORK

5.1 The OMG QVT Relations Language

The MOF 2.0 Query / View / Transformation specification [13] provides a language for declaratively specifying transformations as a set of relations that must hold between models. A relation is defined by two or more domains, and is declared either as *Checkonly*, meaning that the relation is only checked, or *Enforced*, meaning that the model is modified if necessary to satisfy the relation. It is augmented by a *when* clause that specifies under what conditions the relation must hold, and a *where* clause that specifies a condition that must be satisfied by all the model elements participating in the relation.

Our approach provides a solution similar to the *Checkonly* mode of QVT relations. The main difference is our use of *pivot nodes* to define correspondence conditions and the use of cross links. This allows us to use a look up table to match corresponding nodes. Our approach takes advantage of the transformation framework to provide a pragmatic and usable verification technique that can ensure that there are no critical errors in model instances produced by automated transformations.

5.2 Triple Graph Grammars

Triple Graph Grammars [15] can be used to transformations on models as the evolution of a graph by applying graph rules. The evolving graph must comply with a graph schema at all times. This graph schema consists of three parts, one describing the source meta-model, one describing the target meta-model, and one describing a correspondence meta-model which keeps track of correspondences between the other two meta-models. Triple graph grammar rules are declarative, and operational graph grammar rules must be derived from them.

The correspondence meta-model can be used to perform a function similar to the cross links used here. This provides a framework in which a map of corresponding nodes in the instance models can be maintained, and on which the correspondence conditions can be checked. This makes it suitable for our verification approach to be applied.

5.3 Other Verification Approaches

Some ideas on validating model transformations are presented in [7] and [8]. In [8], 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 [7], 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. We focus on the semantic correctness of model transformations, addressing errors introduced due to loss or mis-representation of information during a transformation. It is possible for a transformation to execute completely and produce an output model that satisfies all syntactic rules, but which may still not have accomplished the desired result of porting essential information from the source model to the output model. Our approach is directed at preventing such semantic errors.

Ehrig et. al. [4] study bidirectional transformations as a technique for preserving information across model transformations. They use triple graph grammars to define bidirectional 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.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced an approach for the instance-based verification of model transformations based on the concept of structural correspondence. We have illustrated its use on examples and described informally how the structural correspondence could be specified in a language. We conjecture that the technique is viable and pragmatic for a wide range of transformations, where correctness can indeed be captured in such structural form, but the wider applicability of the approach needs to be verified on industry-grade examples.

Our approach is based on the assumptions that (1) it is relatively easy to specify the correspondence criteria for a transformation, and (2) the correspondence rules are 'complete' in the sense that they cover all the relevant semantic aspects of the transformation. As these properties are always domain- and transformation-specific, it is hard to quantify what it costs to satisfy them. However, we believe it is feasible to integrate such checks into practical transformation applications and, if the specifications are indeed independently developed, they could serve a solid foundation for verifying the results of transformations.

The most important future work in the approach is the full implementation of the correspondence language and the model checker that evaluates the correspondence expressions. Since the correspondence specification does not make any assumptions about the transformation algorithm, it will be possible to plug it into any implementation of the transformation. Checking each execution of the transformation makes the verification tractable, but care must be taken to not create a significant overhead. However, the guarantee provided about the correctness of the output model can prove to be extremely useful in critical scenarios.

The approach described here will suit a majority of *exogenous* transformations [10] where a structural view can be taken to verify correctness, it may also prove to be useful in *endogenous* transformations (where the source and target models belong to the same meta model) such as optimization or refactoring. Cross links are directed links, and can be created within the same meta. Thus they can be used to specify structural correspondence within the same metamodel. Further research is needed to study the applicability and issues (such as possibility of cycles) of using this technique for verifying such transformations .

7. REFERENCES

[1] A. Agrawal, G. Karsai, and 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*, pages 8–15, New York, NY, USA, 2003. ACM Press.

[2] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model Transformations in Practice Workshop Announcement, MoDELS 2005, 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf`.

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

[4] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *Fundamental Approaches to Software Engineering*, pages 72–86. 2007.

[5] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

[6] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[7] J. M. Küster. Systematic validation of model transformations. In *Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004)*, October 2004.

[8] J. M. Küster, R. Heckel, and 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*, pages 145–152, Washington, DC, USA, 2003. IEEE Computer Society.

[9] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

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

[11] A. Narayanan and G. Karsai. Using semantic anchoring to verify behavior preservation in graph transformations. *Electronic Communications of the EASST*, 4(2006), January 2006.

[12] A. Narayanan and G. Karsai. Verifying Model Transformations by Structural Correspondence. In *7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, March 2008.

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

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

[15] 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*, pages 151–163, London, UK, 1995. Springer-Verlag.