Executing a Platform Independent Model of the *UML-to-CSP* Transformation on a Commercial Platform

Pieter Van Gorp, Olaf Muliawan, Anne Keller, Dirk Janssens

{pieter.vangorp,olaf.muliawan,anne.keller,dirk.janssens}@ua.ac.be
Department of Mathematics and Computer Science
University of Antwerp

Abstract. Model-Driven Engineering is a software development method that enables one to model applications at a high level of abstraction and introduce platform specific details automatically by means of model transformations. Similarly, models specified in human-readable visual languages can be mapped automatically onto lower-level languages that enable one to formally derive properties such as termination, deadlock freeness, etc. Since different commercial tools tend to store models in slightly different ways, the transformations that automate mappings between models need to be modeled as well. This enables one to develop transformations that integrate with tools whose metamodels differ slightly from the corresponding language standard. This paper applies MoTMoT to transform a platform independent, human-readable, and visual model of the *AGTIVE UML-to-CSP* transformation into code that integrates with a tool that slightly deviates from the UML 2.0 standard.

1 Introduction

In order to facilitate a comparison of the expressiveness, usability and performance of graph transformation tools, three case studies have been published in preparation of the third international workshop on *Applications of Graph Transformation with Industrial Relevance* (AGTIVE [3]). MoTMoT is a tool that transforms UML models of controlled graph transformations into executable Java code that conforms to the Java Metadata Interface (JMI [5]), an API standard for accessing model repositories. It has been designed to illustrate how several model transformation problems of the Fujaba tool can be solved [8]. The following features make MoTMoT still unique today:

1. MoTMoT is based on a UML compliant implementation of Story Diagrams. Story Diagrams support all constructs that have emerged from decades of research in controlled graph transformation. Therefore, MoTMoT's language may be considered to be expressive. Moreover, the UML 1.5 profile based implementation of this language enables industrial software engineers to specify graph transformations in their modeling tool of preference. Due to the maturity of industrial UML tools, features such as transformation views are available without any investment from the MoTMoT developers.

2. MoTMoT can be applied on OMG's MDA compliant inputs. On the one hand, models residing in a MOF repository can be transformed directly. On the other hand, file integration is supported by relying on the XMI standard.
3. MoTMoT supports reflection. Not only can dynamic type information be extracted from input model elements: since transformations are stored as MOF model elements, their meta-information is available at runtime too.
4. MoTMoT is extensible. More specifically, language constructs can be added to conventional story diagrams by relying on higher-order transformations [9].

As developers and users of the MoTMoT tool, we consider the AGTIVE "Graph Transformation Tools Contest" as an excellent opportunity to present these features to potential users of this tool and to developers of other graph transformation tools. Therefore, we have applied MoTMoT for realizing the transformation from UML activity diagrams to Communicating Sequential Processes (i.e., the *UML-to-CSP* transformation) that has been proposed as the *model transformation benchmark* for the AGTIVE contest.

This paper describes that transformation and is structured as follows: Section 2 briefly introduces the reader to the description of the benchmark in general and more extensively discusses in what industrial scenarios the features of MoTMoT are most relevant. Section 3 presents the diagrams that model the MoTMoT *UML2CSP* transformation class. Section 4 summarizes the lessons one can learn from this case study. Finally, the paper concludes by evaluating the strengths and drawbacks of the MoTMoT approach.

2 UML-to-CSP: General Requirements and Specific Challenges

The UML-to-CSP transformation that is discussed in this paper is based on the problem description of Bisztray et al. [1]. Instead of repeating the relevance of the problem and a sample of input and output models, this section focuses on the challenges that were identified by Bisztray et al. and the additional challenges that are tackled by this paper.

2.1 General Requirements of the Case Study

The problem description refers to four challenges that need to be tackled by any tool-supported solution [1]: first of all, the tool should support the notion of metamodels (i.e., type graphs). Secondly, the rewriting language should support the updating of node attribute values. Thirdly, it should be possible to define rule application conditions in terms of such attribute values. Finally, rewrite rules need to be embeddable in a control flow. The MoTMoT approach described in section 3 of this paper meets al these requirements.

2.2 Optional Requirements of the Case Study

As an optional requirement, the problem description refers to verification support. Due to the expressiveness of story diagrams (e.g. support for recursion), MoTMoT is unable to decide in general whether or not a particular transformation will always terminate. However, MoTMoT does support several syntactical and semantic checks.

First of all, the underlying java compiler reports any node type ambiguities or conflicts that are left in a transformation model. These checks revealed the differences between the specification of the simplified UML 2 metamodel given by Bisztray et al. and that of the commercial modeling tool used to test the discussed MoTMoT transformation. More specifically, the unqualified type names from the simplified metamodel are highly ambiguous (due to the presence of the given metaclasses in more than one package of the UML 2 metamodel). Additionally, the UML 2 metamodel from the case study assignment subtly differs from the standard OMG UML 2.0 metamodel. For example, the simplified metamodel defines the name attribute only in the *ActivityEdge* and *Action* classes whereas this name attribute is actually inherited from a shared superclass in the standard OMG UML 2.0 metamodel.

MoTMoT also supports semantic checks on the control structure of a transformation. For example, it checks whether a sequence of states within a loop body ends in the state where the nodes over which the loop is iterating, are updated.

2.3 Additional Challenges concerning Industrial Relevance

The additional challenge that is covered by the MoTMoT solution presented in this paper concerns the reconciliation of the following two conflicting concerns: on the one hand, it is desirable to model the transformation independently of a particular tool used to produce input UML models. On the other hand, the code generated by MoTMoT should correctly transform UML models produced by an industrial tool that subtly violates the UML 2.0 standard and that was developed independently of MoTMoT and independently of the case study under consideration.

The first concern has been tackled by defining a transformation strictly in terms of standard OMG UML 2.0 metaclasses (making the transformation model platform independent). In the context of the second concern, the transformation was tested with input produced by MagicDraw 10.0, an industrial UML 2.0 tool that deviates slightly from the OMG standard. The vendor of MagicDraw provides libraries to read the XMI 2.1 files produced by MagicDraw 10.0 into a MOF compliant repository [6]. Unfortunately, the content of this repository is not completely OMG UML 2.0 compliant. More specifically, two challenges need to be overcome:

1. The MagicDraw editor does not enable one to create instances of *MergeNode* and *JoinNode*. Instead, the concepts of a merge and join are realized by

relying on the *DecisionNode* and *ForkNode* metaclasses (which are already used for realizing the concepts of a decision and a fork too). This violates some UML 2.0 well-formedness rules (WFRs) but is a given fact that needs to be dealt with when integrating with industrial tools.

2. The inheritance links induced by the MOF 2.0 package merges in the UML 2.0 metamodel definition are realized by MagicDraw-specific metaclasses.

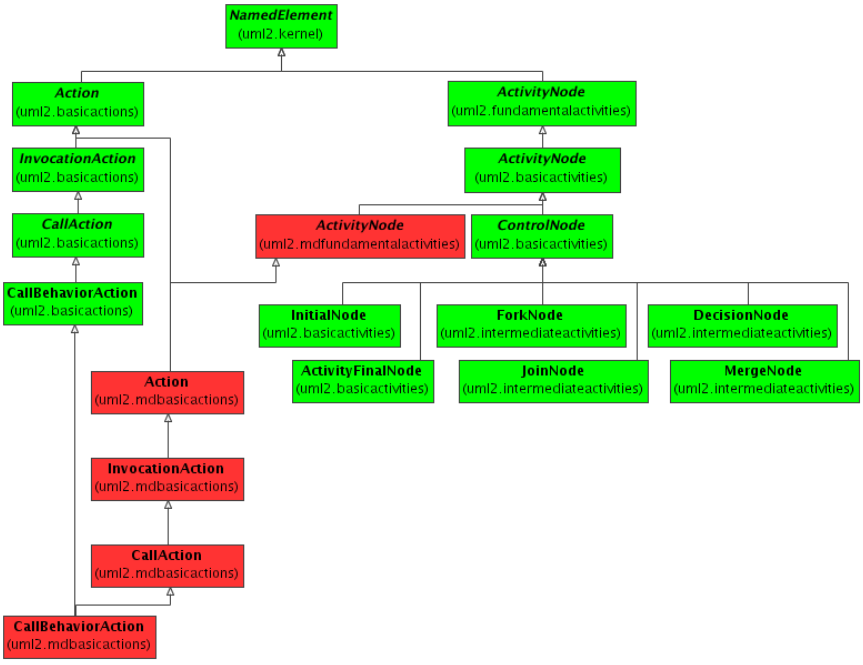


Fig. 1. Metamodel induced by the industrial UML tool used to produce input models.

Figure 1 shows the fragment of the MagicDraw 10.0 metamodel that highlights the concrete integration problem following from the proprietary way in which the inheritance links induced by package merges are realized (challenge 2): instead of providing a specialization link from *uml2.basicactions*'s *Action* class to *uml2.basicactivities*'s *ActivityNode* class, the merge of the "actions" hierarchy with that of the "activities" hierarchy is realized in the MagicDraw-specific *CallBehaviorAction* class from *uml2.mdbasicactions*. Note that all classes in packages with the "md" prefix are MagicDraw-specific, which is emphasized with colors in Figure 1. The next section will illustrate that in order to model the transformation in an elegant manner, one needs to reason about nodes that are both of type *Action* and of type *ActivityNode*. However, the transformation model cannot rely on the *CallBehaviorAction* class from *uml2.mdbasicactions* as this would make the model MagicDraw-specific.

In general, these two issues should illustrate that the use of commercial, off-the-shelf, modeling tools leads to challenges that are not encountered when using "in-house" academic software only. The following sections also tackle these issues to illustrate the flexibility of MoTMoT.

3 MoTMoT: Transformation Model for UML-to-CSP

This section describes the MoTMoT artifacts that have been developed as a solution to the UML-to-CSP case study. Subsection 3.1 discusses the architecture of the solution by modeling the interaction of the generated MoTMoT transformation class with its environment. Subsection 3.2 describes the structure of the transformation class along with its input and output metamodels while subsection 3.3 describes the behavior of the methods that realize the actual mapping from UML elements to CSP constructs. Finally, subsection 3.4 explains what techniques have been applied to ensure the transformation produces correct output even when its input does not exactly conform to the standard UML 2 metamodel.

3.1 Architecture

This subsection introduces the reader to the architecture of the solution by describing which code artifacts are generated and which ones are written by hand. Moreover, the role of the different modeling tools is explained by means of a sample application of the solution.

Figure 2 displays how a UML 1 tool can be used to edit transformation models conforming to the UML profile for Story Diagrams. MoTMoT applies a set of AndroMDA templates to generate a Java implementation from such a transformation model. In the example given, the UML2CSPImpl.java file displayed at the top of the diagram, represents such a generated implementation file. The UML2CSP transformation class is instantiated from a JUnit test, called *UML2CSP_Test.java*. The test reads the input XMI file, produced by a UML 2 tool, into a MOF repository and loads the activity diagram that needs to be transformed by the *UML2CSP* instance. After executing the "transform" method of the transformation instance, the resulting *CspContainer* instance is serialized to XMI again. Finally, an XSL template transforms the abstract syntax instances from the XMI file into expressions in actual CSP syntax.

All case study artifacts are available in the "*samples/uml2csp*" directory of the MoTMoT distribution that can be downloaded from *http://motmot.sf.net/*. In "model.xml", several sample activity diagrams are defined in MagicDraw 10 (i.e., UML 2) format. The "AgtiveSample" diagram is a copy of the sample from the benchmark description. Other activity diagrams have been added to illustrate the generality and robustness of the solution that is presented in this paper. The file called "metamodel.xml" contains the MOF 1 metamodel defining UML 2. The file called "transformation.xml" has been constructed with MagicDaw 9 and contains all UML 1 based Story Diagrams presented in the following sections.

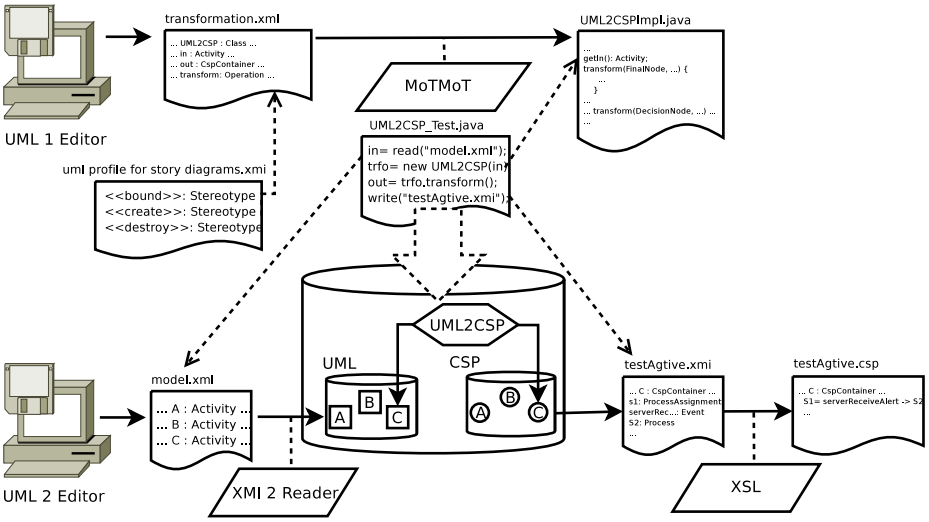


Fig. 2. Architectural model of the Case Study Solution

3.2 Structure: Related Elements and Mapping Responsibilities

This subsection describes how the structure of the transformation model relates to that of the input and output models. The structure of the input metamodel has been discussed in the previous section. The output metamodel is structurally identical to the CSP metamodel from [1] but is made more precise giving explicit names to all association ends (which is required to resolve an ambiguity in the *ProcessExpression* class, due to the two incoming compositions starting from the *BinaryOperator* class). Figure 3 presents the MOF compliant metamodel for the structure of the UML2CSP transformation.

As Figure 3 indicates, each *UML2CSP* instance should have exactly one reference to an *Activity* instance with name *in*. This reference represents the input UML 2 activity diagram that needs to be transformed into a *CspContainer*. Quite simply, this task can be accomplished by calling the only public method of the *UML2CSP* instance: *"transform"*. The behavior of this method is realized by a subclass called *UML2CSPImpl*.

As can be seen from the corresponding stereotypes, the six other *transform* methods in *UML2CSPImpl* realize a model transformation. Each such method has two parameters: the first parameter represents the input UML model element that needs to be transformed while the second parameter represents the *CspContainer* to which generated CSP assignment expressions need to be added. These transformations are strictly *out-place* since they create elements in the output model while leaving the input model unaffected [4]. Moreover, the transformations are exogenous since the input metamodel differs from the output metamodel.

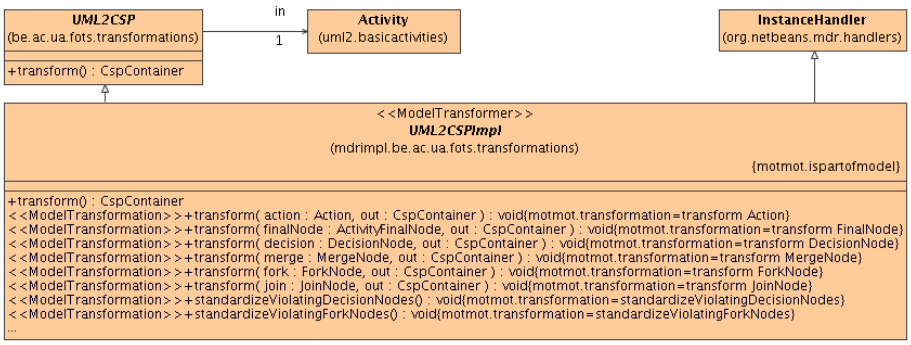


Fig. 3. Structure of the UML2CSP transformation model.

Coming to the technical aspect of defining an abstract method that is overloaded by the six transformation methods, the second challenge following from the industrial constraints (discussed in Section 2.3) comes into play. More specifically, one would want to provide an abstract method *transform(inputElement, outputContainer)* where the first parameter is of type *ActivityNode* and the second parameter is of type *CspContainer*. However, due to the lack of a specialization link from *Action* to *ActivityNode*, the first parameter is typed with the more general *NamedElement* metaclass. This undesirable mismatch between the transformation writer's intention and the required MoTMoT transformation model is a known limitation for which a general solution has not yet been proposed. Instead, it is treated as an implementation aspect that is hidden from diagrams such as the one from Figure 3.2.

The *motmot.transformation* tagged value on each of the transformation methods indicates in which package one can find the Story Diagram modeling the behavior of the method under consideration. The two transformation methods whose name starts with "standardize" contribute to the robustness of *UML2CSP-Impl*. More specifically, they normalize non-standard constructs from the input UML in-place transformations into standard UML constructs by means of in-place transformation steps.

Finally, Figure 3 indicates that *motmot.ispartofmodel* is true for *UML2CSP-Impl*. Additionally, it indicates that this transformation class specializes the *InstanceHandler* interface. This configures MoTMoT such that the objects of type *UML2CSP* can be stored as a model element in a repository too.

3.3 Behavior of the Transformation Methods

This subsection presents the story diagrams that model the behavior of the complete transformation. These story diagrams are presented in the same chronological order as the mapping rules in section 3 of the case study's problem description [1].

Transform Action Node The mapping of an UML Action to a CSP *Process-Assignment* can essentially be realized in one step, that is modeled by the Story Pattern (i.e., "primitive graph transformation rule") shown in Figure 4.

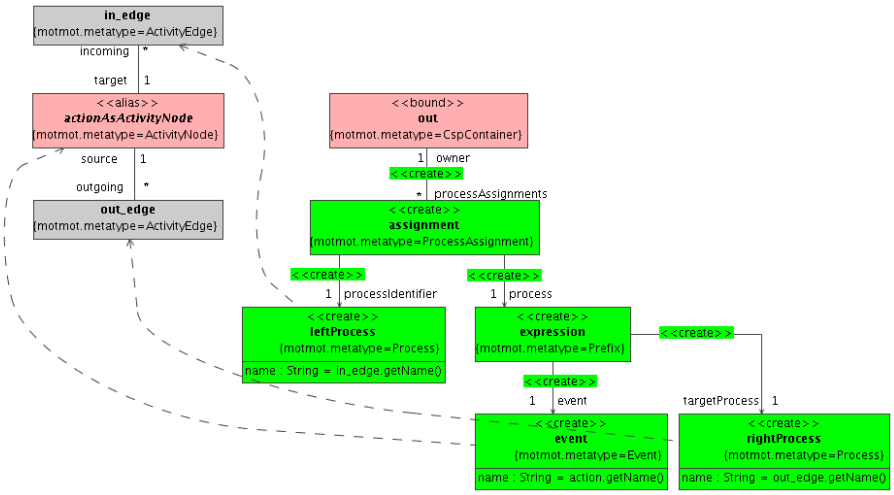


Fig. 4. Story Pattern Transforming Action Nodes.

In the UML profile for Story Diagrams, Story Patterns are encoded using stereotypes and tagged values on classes and associations as they are shown on class diagrams. Classes and associations without a stereotype, or those with the ≪destroy≫ stereotype represent nodes or edges that need to be matched in the host graph. They are thus part of what is commonly known as the left-hand side of the rewrite rule. Elements with a ≪destroy≫ stereotype need to be removed when a match is found. They thus correspond to elements that are part of the left-hand side of the rewrite rule without being part of the right-hand side. Conversely, elements with the ≪create≫ stereotype should be created when a match is found. They thus correspond to elements that are part of the right-hand side of the rewrite rule without being part of the left-hand side. Nodes marked as ≪alias≫ are direct aliases for bound nodes. Aliased nodes can have a more generic type than the nodes they are rebinding.

Node types are specified using the *motmot.metatype* tagged value, node attributes are specified as class attributes and assignments to attribute values are specified as initial values (i.e., using the "=" character). Note that node types are fully qualified within transformation models although the diagrams in this paper only show unqualified type names as values for *motmot.metatype*.

Edges are typed by the pair of association end names. These edge types can be expressed more concisely by using unidirectional association links. This technique not only saves space on Story Diagrams, it also influences the matching process:

at rule application time, the link will be traversed in the direction indicated by the Story Pattern.

Throughout this paper, we apply the following stylistic conventions to improve the readability of Story Patterns: nodes that are bound are colored light-red, nodes that need to be matched are colored gray, nodes (and links) that need to be removed are colored red and nodes (and links) that need to created are colored green. Moreover, UML dependency links (dashed arrows) are used to indicate the source node of the dependency link is generated from the target node of the dependency link. Note that these annotations are applied for documentation purposes only and thus have no effect on the MoTMoT code generator.

Coming back to Figure 4, one can now read this Story Pattern as follows: if the "input" *Action* node has both an input transition *in_edge* as well as an output transition *out_edge*, generate a corresponding CSP assignment by creating a process with the same name as the *in_edge* at the left-hand side of the assignment operator (i.e., as the *processIdentifier* of the assignment) and a prefix expression at the right-hand side of the assignment operator. The event of the prefix expression gets the name of the input *Action* node while the *Process* node at the right-hand side of the prefix expression's arrow operator gets the name of the output transition (i.e., *out_edge.getName()*). Note that the *actionAsActivityNode* and *out* nodes are already bound at rule application time since they are passed as arguments to the transform method: the *out* node corresponds to the second parameter of the transform method, while the *actionAsActivity* node is an alias for this method's first parameter. Because the -*owner*—*processAssignments*- edge carries the ≪create≫ stereotype, the generated CSP assignment expression will be added to the output CSP container *out*. Finally, observe that a complete example of the ≪alias≫ syntax will be presented in the context of the transformation of fork nodes.

Transform Initial Node The transformation of initial nodes does not require a dedicated rule since the expected process assignment is already generated by the other mapping rules. More specifically, all other mapping rules generate assignments to processes whose name is based on that of an incoming transition. If such a transition starts from the initial node, that node will be transformed appropriately too.

Transform Final Node The transformation of a final node requires the creation of a simple assignment from the final node. As Figure 5 shows, the process at the left of the assignment operator gets the name of the input transition of the final node. Syntactically, this name assignment is realized by means of the attribute assignment on the *leftProcess* node. Using the same syntactical construct, the process at the right of the assignment operator gets the name "SKIP". The Story Pattern has a very small application condition: it only requires the final node to have an incoming transition *in_edge*.

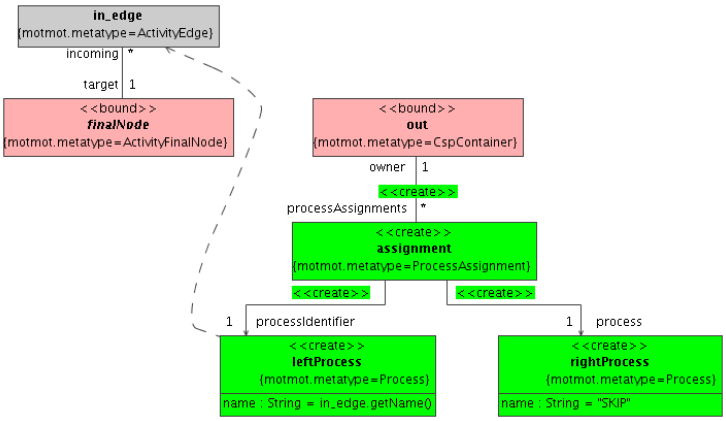


Fig. 5. Story Pattern Transforming Final Nodes.

The Story Diagram that embeds the above Story Pattern in a trivial control flow is not shown.

Transform Decision Node The transformation for decision nodes should generate a CSP assignment expression with at the right-hand side a tree of conditional expressions. An example of such a mapping is shown in Figure 6, which is taken from the case study description [1].

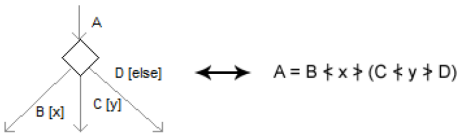


Fig. 6. Example Mapping of a Decision Node.

As can be generalized from this example, the outgoing transition holding a guard called "else" should be mapped to a process that is nested most deeply within the tree of conditional expressions at the right-hand side of the CSP assignment. When providing rewrite rules for all elements involved in the input UML expression (i.e., the input transition, the output transition holding the "else" guard, and finally all other transitions) one requires a means to embed these primitive graph transformation rules in a control flow. In the UML profile for story diagrams, this is realized by decorating an activity diagram with stereotypes that specialize the generic state elements of standard UML and with tagged values that link a state to a Story Pattern or that capture constraints that cannot be expressed elegantly using Story Patterns.

Figure 7 shows the Story Diagram that controls the flow between the different Story Patterns for mapping a decision node to a compound CSP expression. The given story diagram realizes a purely imperative approach that starts with generating the most deeply nested expression ("D" in the example given above), keeps track of the most recently generated expression and iteratively generates conditional expressions that have a process for another outgoing transition (B or C in the example given above) at the left-hand side and the previously generated expression at the right-hand side. As long as outgoing transitions can be found that have not been transformed before, new conditional expressions are generated. Afterwards, the outermost conditional is used as the right-hand side of an assignment expression that is generated for the incoming transition of the decision node.

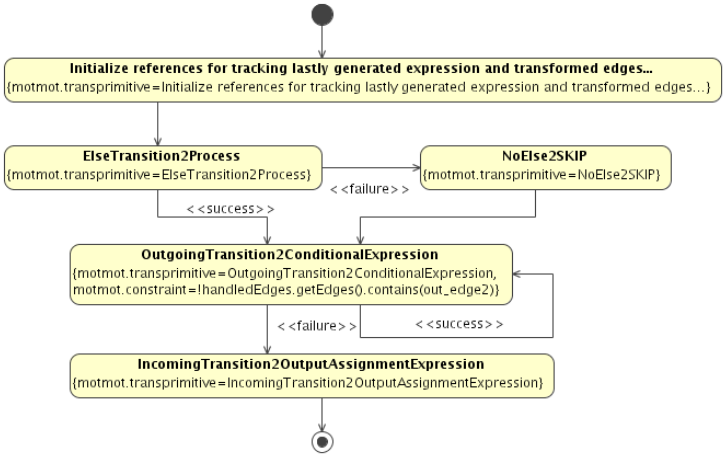


Fig. 7. Story Diagram Transforming Decision Nodes.

The Story Diagram in Figure 7 shows that at first, two auxiliary variables are initialized. From the previous paragraph, one may understand that these two variables are needed to control the rewrite rules:

- one variable keeps track of the most recently generated CSP expression for embedding that expression within the iteratively constructed expression tree,
- another variable keeps track of what transitions have already been transformed.

After initializing these variables in the first state (using a rewrite rule with an empty left-hand side and two new nodes for the helper variables at the right-hand side), the *ElseTransition2Process* Story Pattern creates a process for the transition with a guard named "else". This Story Pattern, shown in Figure 8, contains three bound nodes: the *decision* node is bound, as it represents the first

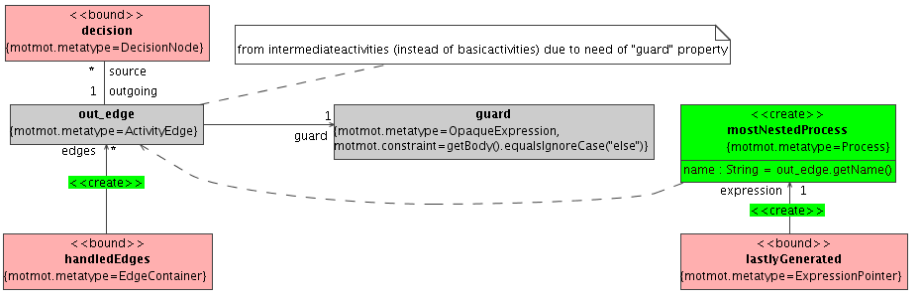


Fig. 8. Story Pattern named *ElseTransition2Process* generates a CSP process expression for the outgoing "ELSE" transition.

argument of the transformation method, and the *handledEdges* and *lastlyGenerated* nodes represent the two helper variables that have been created in the first Story Pattern of the flow for transforming decision nodes. The types of these nodes are *EdgeContainer* and *ExpressionPointer* respectively. These types are defined in the structural transformation model of which a fragment was shown in Figure 3.

As the application condition of the *ElseTransition2Process* Story Pattern, consider the *out_edge* and *guard nodes* (i.e., those nodes that have no stereotype and that are colored in gray). These nodes ensure the rule only fires when the input decision node has an outgoing edge with a guard named "else". The constraint of the guard name is expressed as an attribute value condition on the guard node. If its application condition evaluates to false, the *ElseTransition2Process* is not fired. On the control flow level (i.e., at the level of the Story Diagram shown in Figure 7), this leads to the firing of the ≪failure≫ transition between the *ElseTransition2Process* state and the *NoElse2SKIP* state. Consequently, the *NoElse2SKIP* rule is evaluated. The Story Pattern corresponding to this rule is shown in Figure 9.

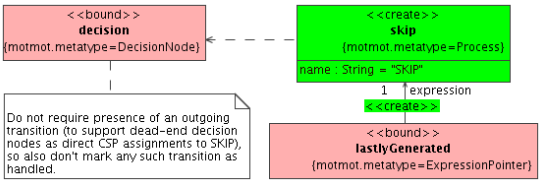


Fig. 9. *NoElse2SKIP*: Story Pattern for mapping a decision node without an outgoing "ELSE" transition to a CSP expression named "SKIP".

The *NoElse2SKIP* rule does not require the presence of an "ELSE" guard. In fact, it does not even require the presence of an outgoing transition. This im-

plies that the transformation for decision nodes should produce correct output in the case no output transitions are present. The *NoElse2SKIP* rule generates a *Process* node with name "SKIP" and directs the *lastlyGenerated* expression pointer to that CSP *Expression* node, so that the subsequent *OutgoingTransition2ConditionalExpression* rule can use the generated SKIP expression as the right-hand side of a conditional expression. Note that in case no outgoing transitions are present, the transformation will continue directly to the *IncomingTransition2OutputAssignmentExpression* rule where the most recently generated expression, which is SKIP in this case, is used as the right-hand side of the generated CSP assignment expression. As such an assignment to SKIP is a reasonable mapping for the exceptional case of a decision node without outgoing transitions, the solution presented here can be considered to be robust.

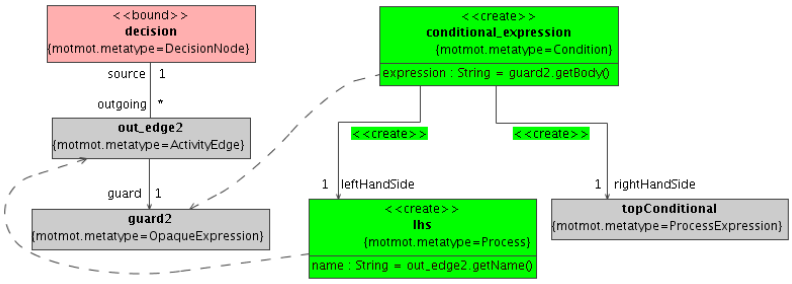


Fig. 10. View on *OutgoingTransition2ConditionalExpression* Story Pattern from the perspective of its core mapping responsibilities.

Since the *OutgoingTransition2ConditionalExpression* Story Pattern involves the creation of quite a number of nodes and edges, we decompose it into two complementary *transformation views*: while Figure 10 shows a diagram highlighting the core mapping responsibilities of the rule, Figure 11 shows a diagram focussing on rewrite nodes and edges that control the termination of the *OutgoingTransition2ConditionalExpression* rule and that connect expressions generated by different rules.

The mapping behavior shown in Figure 10 consists of generating a conditional expression for the given decision node. As visualized by a dotted edge, the expression of this conditional expression is based on the guard of the decision node's output transition. Another dotted edge visualizes that the name of this output transition is used to generate the process at the left-hand side of the conditional expression. The right-hand side of the conditional refers to the conditional that is the top of the expression tree generated by previous rewrite rules.

On Figure 11, one can observe that the *topConditional* node representing this top expression should be matched by navigating the link from the *lastlyGenerated* pointer. Note that this link is redirected once the new conditional expression

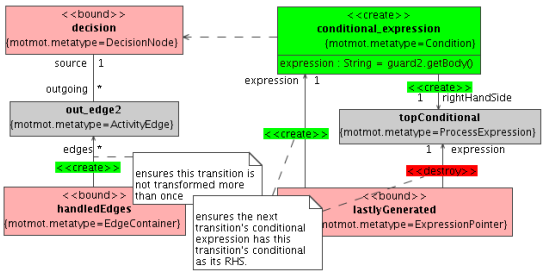


Fig. 11. Transformation view on *OutgoingTransition2ConditionalExpression* Story Pattern from the perspective of rule interaction elements.

is generated. Also note that the *out_ edge2* node is added to the auxiliary *handledEdges* container. The constraint on the Story Diagram state referring to the *OutgoingTransition2ConditionalExpression* pattern (see Figure 7) ensures that an output transition is not transformed more than once into a CSP conditional expression:

$$!handledEdges.getEdges().contains(out\_edge2)$$

This state constraint can also be modeled elegantly by a Negative Application Condition (NAC) but this syntactical construct is not (yet) supported by MoT-MoT.

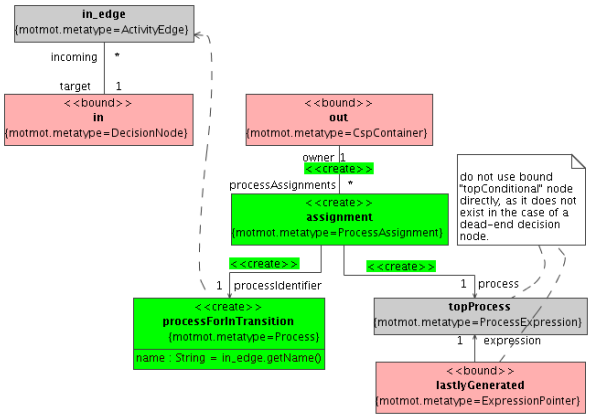


Fig. 12. *IncomingTransition2OutputAssignmentExpression*: Story Pattern mapping the incoming transition of the decision node to the left-hand side of the assignment.

The final Story Pattern controlled by the Story Diagram for transforming decision nodes ensures that the conditional (or SKIP) expression, generated by

the other Story Patterns of the decision node transformation, is embedded in the assignment expression that is added to the global sequence of CSP expressions. More specifically, as Figure 12 shows, the expression tree generated by previously executed transformation rules is looked up by matching the *topProcess* node from the already bound *lastlyGenerated* node. Once found, it is connected to the newly generated *assignment* node by means of a new *process* link. In the CSP metamodel from the case study, this link represents the right-hand side of the assignment. On the left-hand side of the assignment, a process is generated from the input edge of the decision node. Note that the *in* node is an alias of the input *decision* node that is bound as a parameter of the transformation method. This alias is initialized in the transformation's first state and is provided to enable one to reuse the *IncomingTransition2OutputAssignmentExpression* Story Pattern across different Story Diagrams, as will be illustrated when discussing the transformation of fork nodes.

Transform Merge Node One Story Pattern is sufficient to model the behavior of the transformation from merge nodes to the corresponding CSP assignments. Figure 13 shows this Story Pattern, which should be executed as long as matches can be found. Multiple matches of the primitive rewrite rule originate from the *in_edge* node that is iteratively mapped to all transitions that enter the given merge node. As visualized by the dotted edges, the process on the left-hand side of the assignment expression is based on the input transition while the right-hand side is based on the transition that leaves the given merge node.

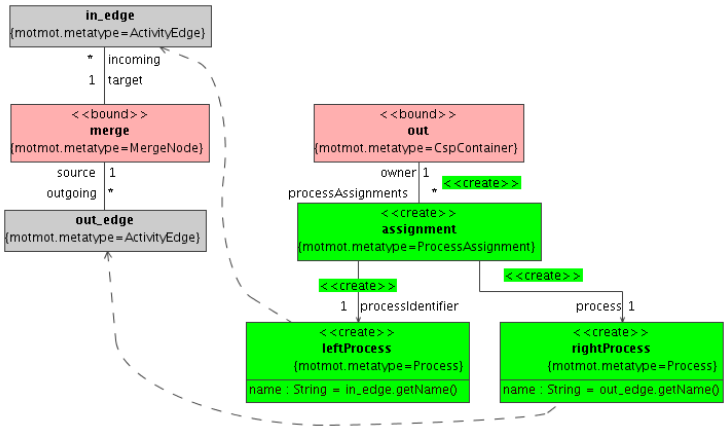


Fig. 13. Story Diagram for transforming Merge Nodes.

The Story Diagram modeling the flow in which the above Story Pattern is embedded, is not shown due to its simplicity: it consists of only one state that is decorated with the ≪loop≫ stereotype.

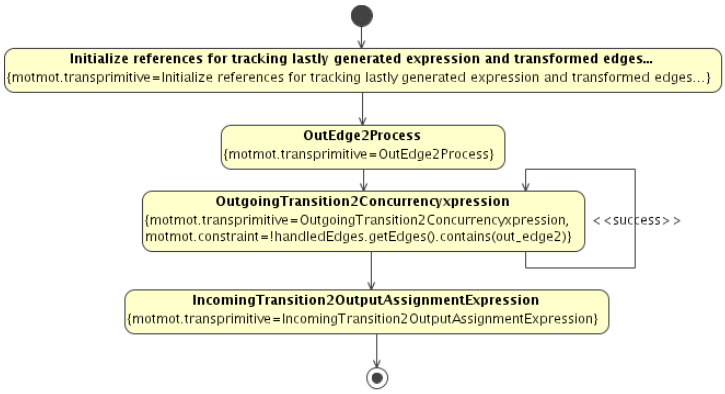


Fig. 14. Story Diagram modeling the control flow of the transformation for Fork Nodes.

Transform Fork Node The control flow of the transformation for fork nodes strongly resembles that of the transformation for decision nodes. The transformation for fork nodes is simpler than that for decision nodes since outgoing transitions have no features that require a treatment different from that of other outgoing transitions. In contrast, recall that transitions leaving a decision node require a special treatment in the case they have a guard labeled "else". Figure 14 models that in the second state of the fork node transformation, an output transition is selected at random and transformed to a CSP process. Afterwards, the transformation transforms all other outgoing transitions into CSP concurrency expressions. Finally, the most recently generated CSP expression is used as the right-hand side of a newly created CSP assignment expression.

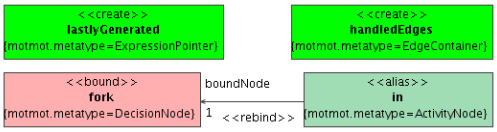


Fig. 15. Story Diagram for initializing auxiliary nodes that support the control flow.

Note that the Story Pattern for the first state binds the same auxiliary nodes as discussed in the context of the decision node transformation discussed before. Moreover, it binds the *in* node as an alias of the *fork* node, in order to reuse a complete Story Pattern in the fourth state of the transformation. Figure 15 illustrates how the UML profile for Story Diagrams supports such node aliasing (including casting) by means of the ≪alias≫ and ≪rebind≫ stereotypes.

Figure 16 models how in the second state, the output transition is selected at random: unlike the corresponding Story Pattern from the decision node trans-

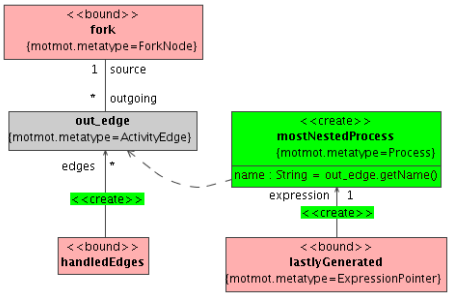


Fig. 16. *OutEdge2Process*: Transformation of a randomly selected output transition.

formation (i.e., *ElseTransition2Process*, displayed in Figure 8), the unbound *out_ edge* node is not constrained by an association with a specific guard node. Apart from that, the meaning of the *OutEdge2Process* pattern is similar to that of the *ElseTransition2Process* pattern.

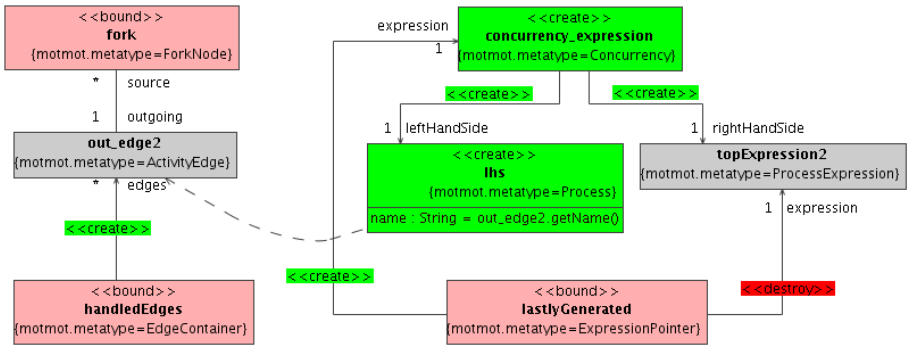


Fig. 17. *OutgoingTransition2ConcurrencyExpression*: Story Pattern transforming an output transition of a fork node into a CSP concurrency expression.

Apart from the type of the topmost expression and the name of the input node, the *OutgoingTransition2ConcurrencyExpression* pattern is identical to the *OutgoingTransition2ConditionalExpression* that has been presented by the two views shown in Figure 10 and 11. Figure 17 shows the complete pattern on one diagram.

As indicated above, the fourth state, called *IncomingTransition2OutputAssignmentExpression*, is reused from the decision node transformation. By creating an alias from the *in* node to the fork node in the first state of this transformation (modeled by the pattern from Figure 15), and by creating a similar alias from the *in* node to the *decision* node in the first state of the decision node trans-

formation (discussed in the context of Figure 7), the name and type differences within the two client Story Diagrams have been encapsulated properly.

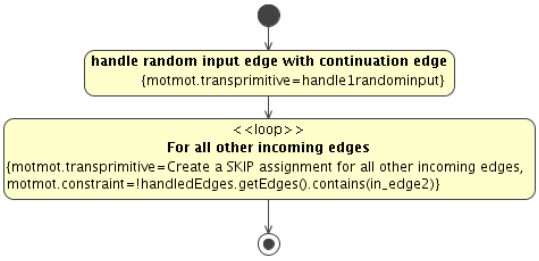


Fig. 18. Story Diagram of the transformation for Join Nodes.

Transform Join Node According to Bisztray et al., the mapping of join nodes involves the most complex transformation. However, from a Story Diagram perspective, the control flow of the decision and merge node transformations is more complex than the one of the join node transformation discussed in this section. In fact, the following Story Diagram is only a minor variation on that for transforming merge nodes.

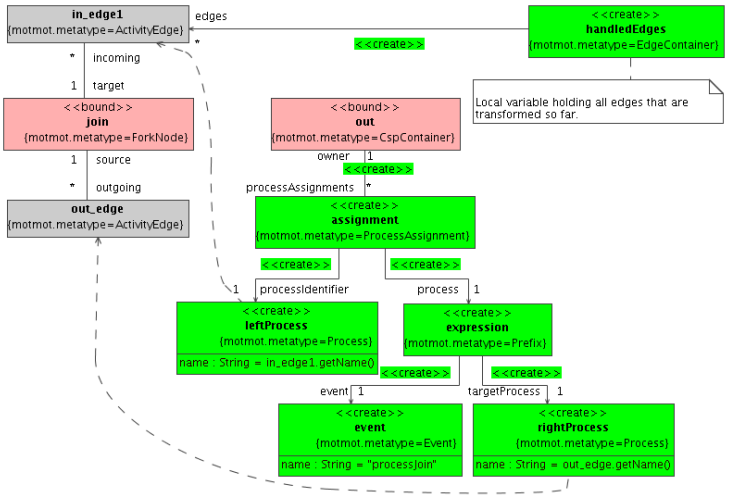


Fig. 19. *handle1randominput*: Story Pattern transforming one input transition to a special CSP assignment.

The Story Diagram for the merge node transformation consists of only one state that is decorated with the loop stereotype. In this state, an assignment is created for an incoming transition. Figure 18 shows that the Story Diagram for join nodes has only one additional state preceding this loop pattern.

Figure 19 shows the Story Pattern for the first state of the transformation. In this state, a transition that enters the input *join* node is selected at random. Once more, a *handledEdges* node is used to keep track of the transitions that have been transformed so far. Join nodes are supposed to have only one output transition. In the rule discussed here, the *out_edge* node is bound to the transition that leaves the input *join* node. When the random input transition and the unique output transition have been found, a newly created CSP expression is added to the output CSP container *out*. The left-hand side of the assignment refers to a process whose name is equal to that of the input transition. The right-hand side refers to a prefix expression consisting of a process whose name is equal to that of the output transition and an event named "processJoin".

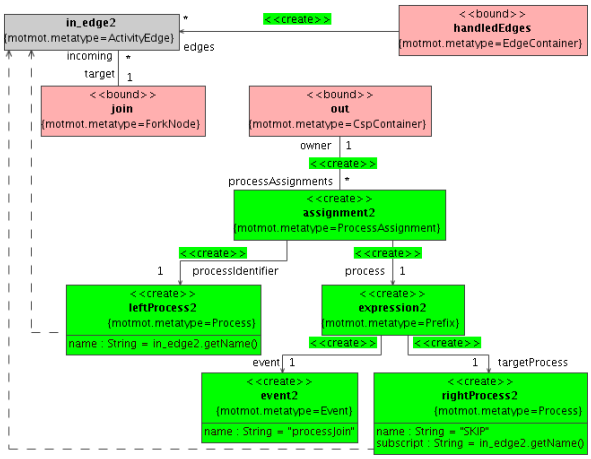


Fig. 20. Story Pattern for transforming all other input transitions.

As Figure 20 shows, the transformation of the other input transitions is modeled by a very similar story pattern. Only one rewrite node differs conceptually from the previously discussed Story Pattern: the *name* and *subscript* values of the process created at the right-hand side of the prefix expression differ from the values assigned in the previous Story Pattern. This follows from the case study's mapping requirements that require a subscripted SKIP statement to be generated for all blocking input transitions. The dotted edges emphasize that in the latter Story Pattern, both processes from the generated CSP assignment are based on the name of the input transition. On the other hand, the dotted edges in Figure 19 are intended to help the reader in verifying that the target process

in the right-hand side of the assignment is generated from the output transition instead.

Transform IN to OUT The Story Diagrams discussed in the previous sections are all based on the overloading of a generic transform method whose first parameter represents the input UML element to be transformed and whose second parameter represents the output CSP container to which the generated CSP expressions need to be added. This section shows how a simple Story Pattern can ensure that this primitive method is executed for all elements of a particular activity diagram.

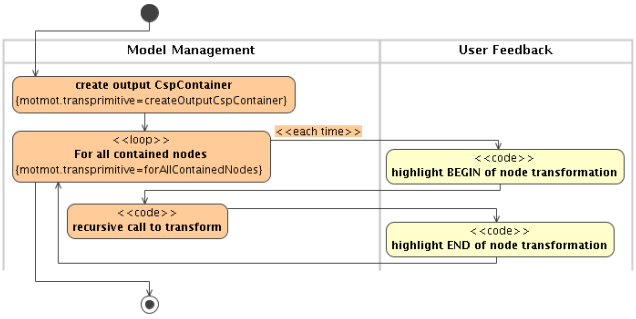


Fig. 21. Top-Level Story Diagram: Transform IN to OUT.

The Story Diagram shown in Figure 21 models the behavior of the *_sdm_transform* (for "story driven modeling" transform) method. The generated implementation of this method realizes the behavior that is expected from the top-level *transform* method discussed in Section 3. As stated, it does not require any parameters in order to generate a *CspContainer* corresponding to the activity diagram referenced by the *in* link of the *UML2CSP* instance. The MoTMoT JMI code generated for the *_sdm_transform* method is not generated into the body of the *transform* method directly since it needs to be surrounded by some exception-handling code in order to satisfy the exception-free signature of the *transform* method.

Figure 21 illustrates how swimlanes can be used to separate the different concerns of a model transformation: the "Model Management" lane contains the states related to the actual mapping while the "User Feedback" lane holds some code states that produce configurable debugging output. Within the left swimlane, the first state creates an empty instance of a CspContainer. The following states will add CSP assignments to this container. The second state iterates over all nodes contained within the input activity diagram. Note that the swimlanes have no semantical effect on the code generator. Therefore, their layout is not constrained and can be optimized in terms of documentation quality.

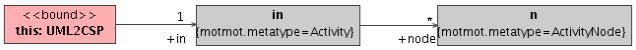


Fig. 22. Story Pattern modeling where the UML activity nodes originate from.

As Figure 22 illustrates, only nodes that are part of the input activity diagram are visited. Although this is accomplished in a trivial manner, such transformation scoping is not easily supported by strict graph grammar-based transformation approaches. The final state of the model management lane contains a method call to the overloaded *transform(in, out)* methods that are modeled by the Story Diagrams discussed in the previous sections.

3.4 Robustness: In-Place Transformation removing non-standard constructs

This section illustrates that MoTMoT not only supports out-place transformations from one or more input model(s) to one or more output model(s): transformations that update an input model in-place can be modeled using Story Diagrams too.

Normalizing UML elements is needed when industrial editors store UML diagrams in a non-standard way. The popular MagicDraw tool, for example, offers in its 10.0 version an editor that instantiates UML *Decision* nodes for representing both decision nodes as well as merge nodes. Similarly, it makes improper use of UML *Fork* nodes by using them to represent join nodes too. This paper presents a solution to the UML2CSP case that handles non-standard fork nodes as well as non-standard decision nodes. The normalization of these non-standard constructs is very similar. Therefore, only the diagrams for the decision node normalization are included here.

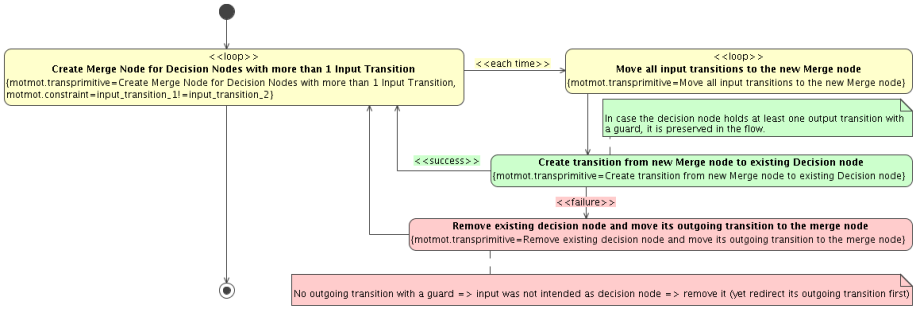


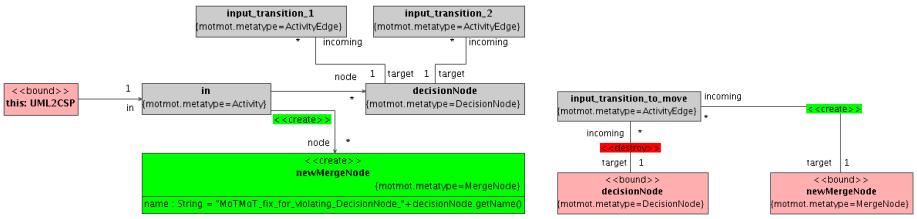
Fig. 23. Story Diagram of the transformation that normalizes non-standard decision nodes.

The Story Diagram shown in Figure 23 models a normalization that is not specific to the MagicDraw 10.0 variant of the UML. More specifically, it transforms any decision node with more than one input transition into a standard counterpart. This involves two cases:

- either the decision node is used with the compound semantics of a decision node and a merge node, by having multiple input transitions as well as multiple output transitions (carrying guards),
- or it is used as a merge node exclusively, by having multiple input transitions but only one output transition.

In both cases, a new merge node element needs to be created and all input transitions of the decision node need to be moved to the new merge node.

In the first state of the transformation, violating decision nodes (i.e., those with more than one input transition) are detected and a corresponding merge node is created for such decision nodes.



(a) Detection of violating decision node and creation of corresponding merge node

(b) Moving a transition to the new merge node

Fig. 24. Story Patterns for the first two states of the normalization for decision nodes.

Figure 24 (a) shows the Story Pattern for this state. Intuitively, the two input transitions are displayed above the violating decision node. The ≪create≫ stereotypes ensure that a newly created merge node is added to the input activity diagram. Note that the violating decision node is not (yet) removed as it may need to be preserved in the input activity diagram. Only in the state that is displayed at the bottom of Figure 23, the decision node can safely be removed. Remark that in contrast to the ≪create≫ and ≪destroy≫ constructs presented in Section 3.3, the ones presented in this section do change the input model. Therefore, the transformation is said to be *in-place* while that of the previous section is *out-place*.

As indicated by the ≪loop≫ and ≪each time≫ stereotypes, the path of states at the right of the Story Diagram shown in Figure 23 is entered for each violating decision node. The first node on this path is decorated with the ≪loop≫ stereotype. This state is responsible for moving all input transitions from the violating decision node to the newly created merge node. Since there

is no ≪each time≫ transition leaving this state, the execution thread remains local to the Story Diagram shown in Figure 24 (b) as long as it can be matched.

The second and third state on the path at the right of the Story Diagram from Figure 23 handle the differences between the two decision node violation scenarios discussed above: the second state normalizes decision nodes that are used at the same time as a decision node and as a merge node, while the third state handles the case where a decision node is abusively used as a merge node only (as in MagicDraw 10.0).

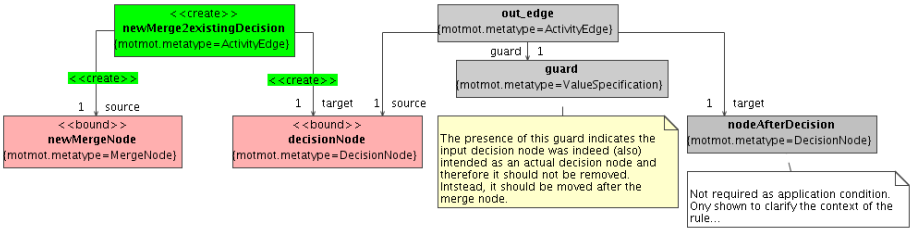


Fig. 25. Story Pattern *prepending* a generated *Merge* node to *Decision* nodes that are used as decision *and* merge nodes at the same time.

Figure 25 shows the Story Pattern for the second state. The gray nodes that are part of its application condition evaluate whether the given decision node has an output transition with a guard. If such a transition is found, the rule concludes the decision node takes an actual decision and therefore should be preserved within the input activity diagram. If such a transition cannot be found, the rule's side effects are not executed and the Story Pattern for normalizing decision nodes triggers the outgoing ≪failure≫ transition. This transfers the transformation in the state where it should replace the decision node, that is obviously used as a merge node, by the generated UML Merge node.

As Figure 26 shows, the Story Pattern for this state deletes the violating decision node from the input activity diagram after redirecting its outgoing transition to the newly generated merge node.

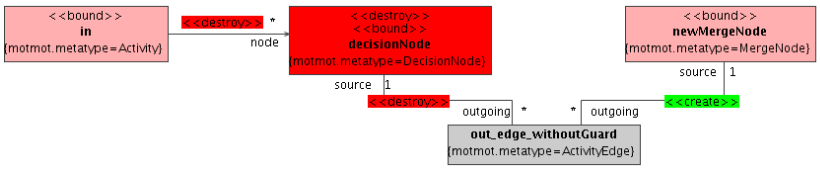


Fig. 26. Story Pattern *replacing* a violating decision node by a generated merge node.

4 Lessons Learned

This section presents a brief overview of the lessons learned from the development of the MoTMoT UML2CSP transformation class.

First of all, the case study illustrates that MoTMoT is effectively usable for the transformation of UML 2 models produced by an industrial modeling tool. Although previous case studies already applied MoTMoT to models conforming to different metamodels (UML 1 [9] and Traceability [7]), a UML 2 case study is an especially convincing means to illustrate that MoTMoT is applicable on any MOF metamodel: the complexity of the UML 2 metamodel did not reveal any scalability problems. More specifically, tests indicate that the performance of the generated MoTMoT code is much better than that of the XMI reader from the CASE tool provider. Apart from this evidence of scalability, the case study solution deals with the MOF 2 package merges that are frequently applied in the standard UML 2 metamodel. Even with a proprietary realization of these package merges at the execution level of the input CASE tool's XMI reader, it appears feasible to model the transformation itself in a standard compliant manner.

Secondly, the case study reinforces our confidence in the quality of the UML profile for Story Diagrams. On the one hand, the expressiveness of controlled graph transformation is clearly illustrated and the elaborate descriptions of the Story Diagrams and Story Patterns may be a useful tutorial supporting the adoption of the formalism in industry. On the other hand, the presented transformation model involves several design choices that heavily influence the readability, compactness, and reusability of the diagrams shown in this paper. For example, several ≪loop≫ constructs can be refactored into combinations of ≪success≫ and ≪failure≫ links. Similarly, some complex Story Patterns can be decomposed into two or more sequentially executed Story Patterns. According to the modeling style of previous publications on Story Driven Modeling [2], the complexity of the Story Pattern from Figure 17 would be tackled by distributing its responsibilities across two states: a first state would take care of the actual mapping from input to output element while a second state would establish all connections that relate to the control flow of the transformation. However, the presented solution deliberately merges such sequences into one state. The complexity of the Story Patterns is managed by means of "views". This approach has several advantages: first of all, introducing a sequential execution order where it is not required can be confusing. In terms of the example from Figure 17, there is no reason why one would first take care of the model mapping and only then create control flow-related elements. Secondly, when applying transformation views, one can investigate more easily the complete set of edges that relate to one rewrite node in a coarse-grained step within the transformation process.

Finally, the *UML2CSP* transformation model is the first large example in which the *reuse* of Story *Patterns* is illustrated. Approaches relying on calls to complete Story *Diagrams* force transformation writers to define some artificial methods that have only one state. Instead, the *IncomingTransition2OutputAssignmentExpression* Story Pattern (cfr. Figure 12) is used directly from the

Story Diagram for transforming decision nodes (cfr. Figure 7) and from that for transforming fork nodes (cfr. Figure 14). Since no method call is involved, the renaming of variable names needs to be tackled explicitly. To this aim, this paper used the ≪alias≫ construct. This construct is not restricted to the renaming of node variables: it supports a type-cast operation too.

Conclusions

The UML-to-CSP case study reveals the following strengths and drawbacks in the MoTMoT approach.

First of all, a key benefit of MoTMoT is the visual nature of its modeling language. Throughout this paper, we have illustrated that a proper use of colors and specific layout patterns can significantly improve the readability of transformation models. While these colors may have no specific meaning when learning story diagrams, they decrease the time to read a Story Pattern already after a very short learning period. On the other hand, the layout patterns applied throughout this paper have no relation to the transformation language semantics. Therefore, these patterns can be optimized completely in terms of the input and output modeling languages. For example, in most Story Patterns presented in this paper, the input transition are displayed above activity nodes while output transitions are displayed below. The ability to exploit 2D layout characteristics to improve the mental mapping of a rewrite rule to the most conventional modeling style (putting input transitions on top of nodes in activity diagrams) is a key advantage of visual graph transformation languages such as Story Diagrams.

A second strength of MoTMoT is its conformance with OMG's MDA standards: not only can the *UML2CSP* transformation model itself be edited with any UML 1 compliant modeling tool, the code generated from that model also consumes standard UML 2 models and produces CSP models that conform to a MOF based metamodel. Moreover, Section 3.4 illustrated how models that do not conform to the standard metamodels of their modeling languages can be normalized in-place to conforming versions.

The third and final advantage of MoTMoT that is presented in this paper is its extensibility. This strength is illustrated by the fact that the new ≪alias≫ construct (that was introduced while completing our submission to the *UML-to-CSP* case study) is realized again as a normalizing (higher-order) transformation from the extended profile for Story Diagrams towards the original profile.

Unfortunately, but also obviously, the MoTMoT approach has some drawbacks and limitations.

First of all, a drawback of the UML profile for Story Diagrams is that it does not support the embedded representation of Story Patterns within Story Diagrams. The Fujaba editor does support such a visualization. Similarly, the Fujaba editor provides more advanced "auto-completion" features. Therefore, it may be worthwhile to realize a mapping between MoTMoT's UML profile and Fujaba's metamodel for Story Diagrams.

Secondly, MoTMoT's support for modeling transformations in a standard-compliant, platform-independent, manner is limited. As discussed in Section 3.2, the *UML2CSP* transformation class presented in this paper would have a method *transform(inputElement, outputContainer)* with different parameter types when one would model the transformation completely independent of the target modeling platform. On the one hand, the effect of this concrete problem is rather limited, as the signature can be changed very locally and the effect of its change is managed properly by the type-checks of the underlying Java compiler. On the other hand, one could overcome this limitation by defining a tagged value on metaclasses that have a non-standard realization. In that case, the behavioral models (i.e., the signatures of transformation methods, the Story Diagrams and the Story Patterns) of the transformation would be completely independent. The platform specific concern would be modeled in the structural part of the transformation model that represents the input metamodel.

References

1. Dénes Bisztray, Karsten Ehrig, and Reiko Heckel. Case study: UML to CSP transformation. Technical Report ?, University of Leicester, UK, 2007.
2. Ira Diethelm, Leif Geiger, and Albert Zündorf. Systematic story driven modeling, a case study (Paderborn shuttle system). In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE'04*, Edinburgh, Scottland, 2004.
3. Gabriele Taentzer and Arend Rensink. AGTIVE 2007 Tool Contest Overview. http://www.informatik.uni-marburg.de/ swt/agtive-contest/, 26 July 2007.
4. Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation*, 2005.
5. Sun Microsystems. Java Metadata Interface Specification, June 2002. document ID JSR-40.
6. No Magic, Inc. MagicDraw technology - XMI 2.1 reader for Netbeans MDR. http://magicdraw.com/main.php?ts=navig&cmd_show=1&menu=technology, 2007.
7. Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Traceability and Fine-Grained Constraints in Interactive Inconsistency Management. In Tor Neple, Jon Oldevik, and Jan Aagedal, editors, *Second ECMDA Traceability Workshop*, 10 July 2006.
8. Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation*, 127(3):5–16, 2004.
9. Pieter Van Gorp, Hans Schippers, and Dirk Janssens. Copying Subgraphs within Model Repositories. In Roberto Bruni and Dániel Varró, editors, *Fifth International Workshop on Graph Transformation and Visual Modeling Techniques*, Electronic Notes in Theoretical Computer Science, pages 127–139, Vienna, Austria, 1 April 2006. Elsevier.