Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools

Dániel Varró[1], Márk Asztalos[1], Dénes Bisztray[2], Artur Boronat[2], Duc-Hanh Dang[3], Rubino Geiß[4], Joel Greenyer[5], Pieter Van Gorp[6], Ole Kniemeyer[7], Anantha Narayanan[8], Edgars Rencis[9], and Erhard Weinell[10]

[1] Budapest University of Technology and Economics, Hungary, `varro@mit.bme.hu,asztalos@aut.bme.hu`
[2] Leicester University, UK, {`dab24,aboronat`}`@mcs.le.ac.uk`
[3] Universität Bremen, Germany, `hanhdd@informatik.uni-bremen.de`
[4] Universität Karlsruhe, Germany, `rubino@ipd.info.uni-karlsruhe.de`
[5] University of Paderborn, `jgreen@uni-paderborn.de`
[6] University of Antwerp, Belgium, `pieter.vangorp@ua.ac.be`
[7] BTU Cottbus, Germany, `okn@informatik.tu-cottbus.de`
[8] Vanderbilt University, TN, USA, `ananth@isis.vanderbilt.edu`
[9] University of Latvia, Latvia, `Edgars.Rencis@lumii.lv`
[10] RWTH Aachen University of Technology, Germany, `Weinell@cs.rwth-aachen.de`

Abstract. Graph transformation provides an intuitive mechanism for capturing model transformations. In the current paper, we investigate and compare various graph transformation tools using a compact practical model transformation case study carried out as part of the AGTIVE 2007 Tool Contest [22]. The aim of this case study is to generate formal CSP processes from high-level UML activity diagrams, which enables to carry out mathematical analysis of the system under design.

1 Introduction

Graph transformation provides an intuitive graphical mechanism for capturing model transformations. Many tools have been developed in the past which implemented different graph transformation principles and introduced new extensions to address specific practical requirements. For example, some tools allow to specify a control structure over their transformation rules whereas others remain purely declarative. Also, different tools provide a different degree of expressive power in what kind of graph structures and attribute values they can handle. In the AGTIVE Tool Contest [22], 17 different tools participated and competed on problems of different nature in order to document and classify their strengths and weaknesses.

This case study represents a typical (exogenous) model-to-model transformation from UML activity diagrams [21] to Communicating Sequential Processes [17]. As the de-facto standard for software design, UML [21] activity diagrams are used to describe low level behavior of software components or to represent business-level workflows. In both cases, verification of the behavior can be important to guarantee the quality of service for the components. The purpose of verification can run from a simple liveness or termination check to the verification of refinement between model instances of different levels of abstraction. To verify any aspect of behavior, the activity diagrams have to be provided with a formal semantics. We are using CSP as a semantic domain, defining the mapping from activity diagram to CSP by means of graph transformation.

In this case study, the transformation tools shall support metamodel-based transformation or any equivalent notion of type graphs. Also, support for attribute handling is required, the various names and properties of elements should be dealt with. The ability to define any kind of control structure for rule application and attribute conditions may be an important issue to guarantee that the transformation is deterministic or to improve its performance. However, the transformation problem also gives space to purely declarative solutions. Due to the large variety of solutions, these solutions will now be compared based upon the model transformation-specific features provided by their corresponding tools.

The rest of the paper is structured as follows. Section 2 provides a brief introduction to the model transformation problem serving as this case study. In Sec. 3, we discuss which criteria are important for specifying and executing transformations of UML models to CSP. Moreover, we overview what kind of tools are used to perform the case study, and in what dimensions the solutions differ from each other. Section 4 presents a one-page summary for each individual solution. Finally, Section 5 provides a high-level comparison of the solutions.

2 Case Study "UML to CSP Transformation"

2.1 Metamodels

First, we introduce the source and target metamodels of the UML2CSP problem. A metamodel formalizes the abstract syntax of a modeling language in the form of UML class diagrams. Classes of the metamodel capture the main concepts of the language together with its attributes. The interrelation of such concepts are captured by associations. Finally, classes can be arranged into a generalization (inheritance) hierarchy.

UML Activity Metamodel. The source language is captured by a simplified metamodel for activity diagrams based on [21] (shown in Fig. 1).

Figure 2 shows a simple example activity diagram (taken from [16]) containing two *ActivityEdges* which connect an *InitialNode* with an *Action* and an *Action* with a *FinalNode*. The object diagram on the right shows how this concrete syntax is represented according to the metamodel shown in Fig. 1.

CSP metamodel. The metamodel for CSP, as far as required for the case study, is shown in Figure 3. A *Process* is the behavior pattern of an object with an alphabet of a limited set of events. Processes are defined using recursive process equations (*ProcessAssignment*) with guarded expressions.

The syntax of the process equations is the following.

$$P ::= F \mid event \rightarrow E \mid E \parallel F \mid E \setminus F \mid E \not< b \not> F \mid SKIP \mid STOP$$

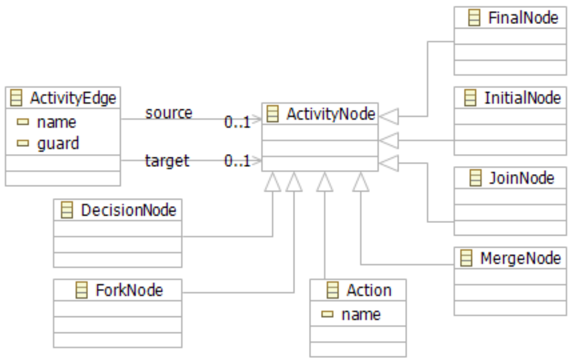


Fig. 1. Activity Diagram Metamodel

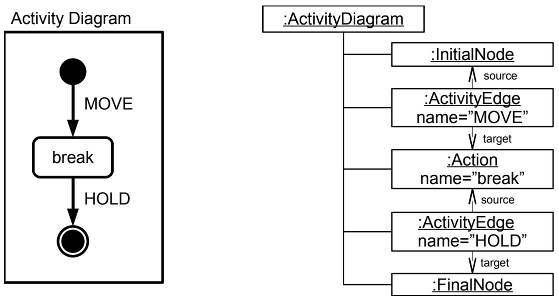


Fig. 2. Simple Activity Model (Concrete and abstract syntax)

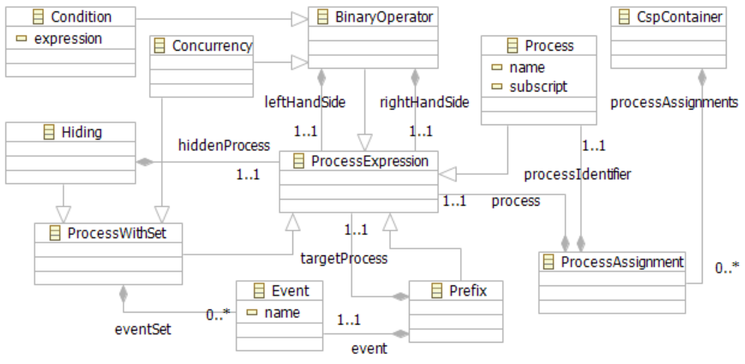


Fig. 3. CSP Metamodel

The abstract class *ProcessExpression* represents a guarded expression. It can be either a simple *Process* P, a *Prefix* operator, a *BinaryOperator* combining two expressions or can be associated with a set of events (*ProcessWithSet*).

The interpretation of the process expressions is as follows. The *Prefix* operator $x \rightarrow E$ performs an *Event* x and then behaves like expression E. If E and F are expressions, *Concurrency* yields their synchronous parallel composition $E \parallel F$ (*performing* E *and* F *simultaneously by synchronizing of shared events*). According to [17], the operator $E \nless b \ngtr F$ is a *Condition* operator, which means, if the boolean *expression* b is true then it behaves like E, else it behaves like F (*if* b *then* E *else* F). If F is a set of *Events* and E is an expression, *Hiding* $E \setminus F$ behaves like E except that all occurrences of events in F are hidden. Finally $SKIP$ represents successful termination, while the $STOP$ process is a deadlock.

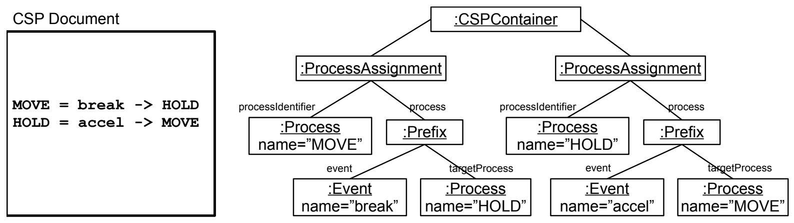


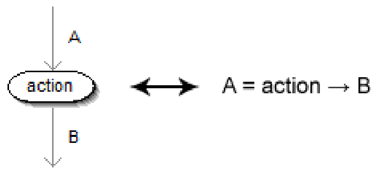
Fig. 4. Simple CSP Model (Concrete and abstract syntax)

Figure 4 (from [16]) shows an example CSP document containing two process assignments. The object diagram on the right shows how the abstract syntax graph of the two statements is built up according to the CSP metamodel show in Fig. 4. In particular, note that there is a Process object for every occurrence of a process in the CSP text.

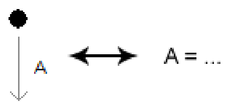
2.2 Overview of the transformation

In this section, we provide an overview of the transformation by showing intuitive correspondences between UML and CSP models. The idea behind the mapping is to relate an Edge in the activity diagram to a Process in CSP. The correspondences are the followings.

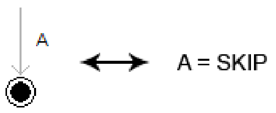
1. An *ActivityEdge* corresponds to a *ProcessIdentifier* while an *Action* to an *Event*. Without loss of generality we restrict Action nodes to have only one incoming and one outgoing edge.



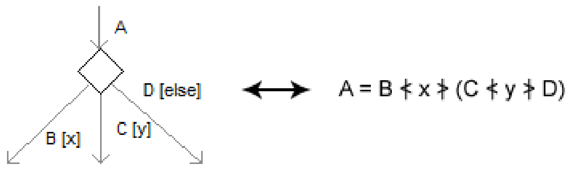
2. *InitialNode* corresponds to the first process assignment.



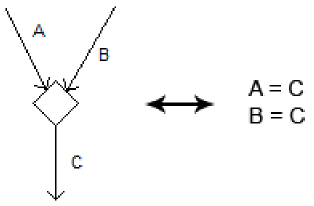
3. An *FinalNode* is a successful termination, thus it corresponds to a $SKIP$ process.



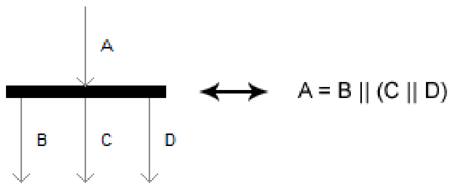
4. A *DecisionNode* corresponds to embedded *Condition* operators with the *guards* as their condition *expressions*.



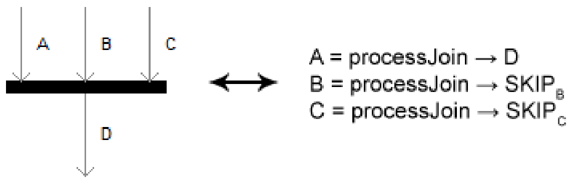
   Note that this correspondence, which creates non-determinism at the syntactic level, leads to semantically equivalent processes. According to [21], *the order in which guards are evaluated is undefined* and *the modeler should arrange that each token only be chosen to traverse one outgoing edge, otherwise there will be race conditions among the outgoing edges*. Hence, if guard conditions are disjoint, syntactically different nestings are semantically equivalent.
5. The *MergeNode* is mapped to an equation identifying the processes corresponding to the two incoming edges.



6. The *ForkNode* corresponds to the *Concurrency* binary operator. Since $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$, the different possible matches are equivalent



7. *JoinNode* represent the most complex cases. Before describing the mapping, we discuss some observations. If in an activity diagram the names of Action nodes are unique, the intersection of the alphabets of the corresponding processes is empty. This is partly intended because in this way the processes will not get stuck while waiting for some random other process that accidentally has events with similarly names. On the other hand we need synchronization points in order to implement the joining of processes. Thus we add an event *processJoin* to the alphabet of every participating processes. Since events that are in the alphabets of all participating processes require simultaneous participation, this fact is used to join concurrent processes by blocking them until they can perform the synchronization event.



   In the concrete mapping the first edge that meets the *JoinNode* is chosen to carry the continuation process, while the others terminate in a $SKIP$. The choice of the first node to be processed in a *JoinNode* is arbitrary, thus we can create multiple, but semantically (i.e. trace, failure and divergence) equivalent set of CSP expressions.

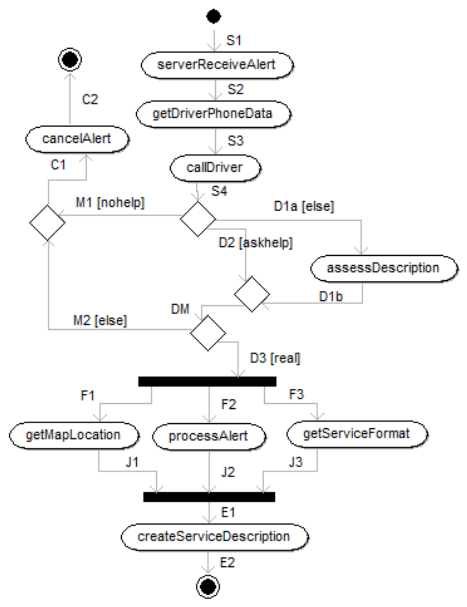
A sample activity diagram and its CSP equivalent (up to process equivalence) is presented in Fig. 5, which was used as a test case for validating the solutions.

The scenario captured by the UML activity diagram describes an autonomous service reacting to an alert issued in case of a car accident. First the driver's cell phone is called to ask if any help is required. If the alert is confirmed by the driver, then the location of the accident (and other service-specific parameters) are sent to the appropriate service provider (e.g. ambulance or tow truck service). This automotive case study is taken from the SENSORIA European Project [25].

2.3 Challenges for the Approach

In this case study, transformation tools should support metamodels or any equivalent notion of type graphs for model management. Metamodels (or type graphs) should also provide support for attribute handling. Control structures and control conditions may provide significant help in specifying the transformation, although their use is not explicitly required.

While performance was not the critical aspect for this case study, it is very important from a practical point of view that a transformation should be executed preferably in a few seconds so that transformation designers may immediately observe and validate the result of their transformation.



$$S1 = serverReceiveAlert \rightarrow S2$$
$$S2 = getDriverPhoneData \rightarrow S3$$
$$S3 = callDriver \rightarrow S4$$

$$S4 = M1 \not\prec nohelp \not\succ (D2 \not\prec askhelp \not\succ D1a)$$
$$M1 = C1$$
$$D2 = DM$$
$$D1a = assessDescription \rightarrow D1b$$
$$D1b = DM$$
$$DM = D3 \not\prec real \not\succ M2$$
$$M2 = C1$$
$$C1 = cancelAlert \rightarrow C2$$
$$C2 = SKIP$$
$$D3 = F1 \mid\mid F2 \mid\mid F3$$
$$F1 = getMapLocation \rightarrow J1$$
$$F2 = processAlert \rightarrow J2$$
$$F3 = getServiceFormat \rightarrow J3$$
$$J1 = processJoin \rightarrow E1$$
$$J2 = processJoin \rightarrow SKIP$$
$$J3 = processJoin \rightarrow SKIP$$
$$E1 = createServiceDescription \rightarrow E2$$
$$E2 = SKIP$$

Fig. 5. Sample source and target models

Finally, the use of some validation techniques is also desirable, although not required. More specifically, termination and determinism up to process equivalence are probably the most relevant questions to be verified for this transformation.

3 Overview on Solutions

In this case study, our primary focus was put on assessing the *expressiveness of different transformation languages* including the expressiveness of the *rule language* itself, as well as the richness of *control structures*, which aim at restricting the applicability of the transformation rules.

Furthermore, solution providers also presented those *advanced features* of their tools, which significantly improved their productivity when creating the solutions. These features included *advanced graphical user interface functionality* (e.g. graphical rule editors) as well as *advanced model transformation features* (such as higher-order transformations, or bidirectionality). Various solution providers highlighted *analysis capabilities* of their tools to pinpoint flaws in the models or the transformation itself. Finally, most of the solutions relied upon an *advanced underlying metamodeling framework* supporting model manipulation and language design.

While the actual solutions were quite different, we identified *common subproblems for all solutions*. One subproblem is how different solutions prevent the applicability of a transformation rule on the same match multiple times. This is a typical problem in many model transformations, thus it offers some comparison specific to the graph transformation problem itself. Another interesting case is the proper handling of outgoing *ActivityEdges* on *Decision* and *ForkNodes*, since in the CSP domain, the outgoing *ActivityEdge* handled last would cause the recursive nesting of *Condition*/*Concurrency*-expressions to be ended. In the current paper, we will put more emphasis on the former subproblem as all the different solutions demonstrate this issue in a compact way.

The UML2CSP case study has been solved altogether by 11 tools, which are categorized (and then presented) below by the overall nature (strategy) of the solution.

– *Pure GT solutions.* Some solutions (like Tiger/EMF in Sec. 4.1 and TGGs in Sec. 4.2) build purely upon core graph transformation formalism with implicit (or minimal) control structure. These solutions demonstrate how model transformations can be formulated in a purely declarative way using graph transformation.
– *Solutions with control structures.* However, most of the solutions rely upon the use of some control structures to restrict the non-determinism of transformations. The underlying tools offer either some *textual language* (like in case of PROGRES in Sec. 4.3 or GrGen.NET in Sec. 4.4), or some *graphical syntax*, typically with a UML flavour as in case of VMTS (Sec. 4.6), USE (Sec. 4.7), MoTMoT (Sec. 4.8), and GrTP (Sec. 4.9). In GReAT (Sec. 4.5), the control language is mostly dataflow based.
– *Solutions with a host framework / language.* There are two solutions, which rely upon a host framework not particularly designed for model / graph transformations. In the solution developed in MOMENT2-GT (Sec. 4.10), the transformation rules are translated into the Maude rewriting framework [11], while XL (Sec. 4.11) uses native Java as its control language.

4 Solutions

In this section, each solution individually describes the principles of the approach with the main settings and highlights of the used tool. Each solution is demonstrated by an example, which gives a brief insight to the look-and-feel of the transformation language or the tool itself.

4.1 Solution using Tiger EMF Transformer

The solution is a set of graph transformation rules on EMF [12] models, that are designed using the Visual Editor of Tiger EMF Transformer [26], and run in the

Eclipse development platform. The production rules are defined by rule graphs, namely a left-hand side (LHS), a right-hand side (RHS) and possible negative application conditions (NACs).

Transformation Mechanics. As the case study is presented with a list of intuitive correspondences between the source and target metamodels, the transformation rules are also in groups that resemble the correspondences. These rules are the implementation of the transformation concept introduced in [10].

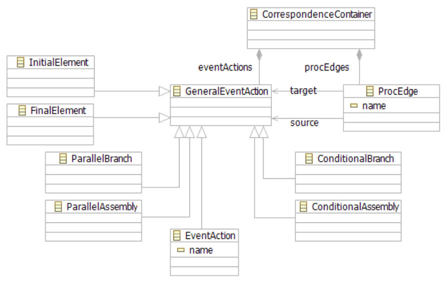


Fig. 6. Correspondence Metamodel

The rules also use a third, correspondence metamodel shown in Figure 6. For example the corresponding element between an *Edge* of activity diagram and a *Process* of CSP is the *ProcEdge*. The role of this metamodel is similar to correspondence structures used by Triple Graph Grammar [23] rules.

The benefits of using it are twofold. Firstly, it is used to mark the procession of the nodes in the source model. This way, we refrain from deleting any nodes in our rules. Secondly, all the possible NACs refer to elements from the correspondence model, as these elements are created during the transformation, never deleted. In [10] it is shown that these properties are important to make the transformation compositional.

The transformation consists of 11 rules within 7 groups. To show the background mechanics of the transformation, we introduce one rule in detail, the transformation of an *Action*.

Rule Descriptions The transformation starts with the application of the **Initial** rule that transforms the *InitialNode*. The rule processes the only outgoing edge from the *InitialNode* by creating an empty process assignment, and also a corresponding *ProcEdge*, to track that this edge has been processed. The NAC guarantees that no edge has been processed in the model before.

The **Action Rule** depicted in Figure 7 is the essential transformation rule. The definition of the previously processed edge A is completed, and the new edge C is indicated as processed and an empty definition is opened for it. The definition of process A is a prefix operator from *Event* B to target process C.

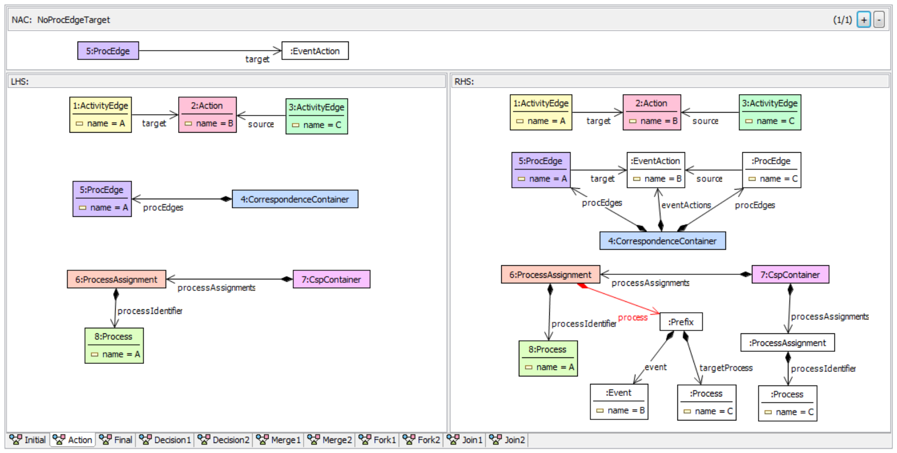


Fig. 7. Action Rule

Remaining Rules The **Final** rule processes a *FinalNode* and fills the previous empty process definition with a *SKIP* process. **Merge1** and **2** rules process the *MergeNode* and connect the empty process definitions with the process that corresponds to the outgoing edge of the *MergeNode*. **Fork1** and **2** transform the *ForkNode* to a 2-regular tree of *Concurrency* operators the similar way the *DecisionNode* tree is built. And finally **Join1** and **2** process the *JoinNode* by creating the synchronization event and related processes. The entire set of rules is available in [9].

4.2 Solution using Triple Graph Grammars

The transformation from UML Activity Diagrams to CSP given in this case study is a typical application for Triple Graph Grammars (TGGs) [23]. Their main advantage over other (single) graph rewriting approaches is that TGG rules reflect the relation between model patterns. This relation can be interpreted in different ways: To translate models in a forward or backward direction, or to maintain their consistency.

The representation of corresponding model patterns is furthermore quite intuitive. In fact, the transformation rules can often be derived from examples in the transformation specification, such as given for this case study. Fig. 8 shows how rules are derived from the information that an *ActivityEdge* corresponds to a *Process* in a *ProcessAssignment* (i) and that an *Action* relates to a *Prefix-Expression* (ii), compare the specification given in Sec. 2. The elements marked green (and with ++) are those essentially related by a rule. Their relationship holds when a certain structural context (b/w nodes) is given. This context implies dependencies to other rules. For example, the second rule in Fig. 8 requires two *ActivityEdges* to previously be translated by another rule, e.g. the rule in (i). This principle of TGGs is very similar to the declarative languages specified by the OMG's model transformation standard QVT [20] as pointed out in [15].

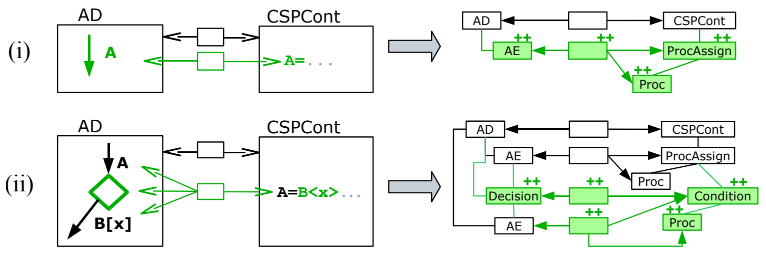


Fig. 8. Two TGG rules derived from specification examples

Our solution is realized with a TGG interpreter plug-in for Eclipse which transforms EMF models. The rules can be modeled in a graphical editor generated with GMF. For details of our solution, refer to [16]. For this example and many others, the interpreter performs very well. For bigger models, the matching algorithm could still be improved, or we would want to compile the rules into executable code. This is done in other TGG transformation engines, like implemented in Fujaba [28] or MOFLON [4].

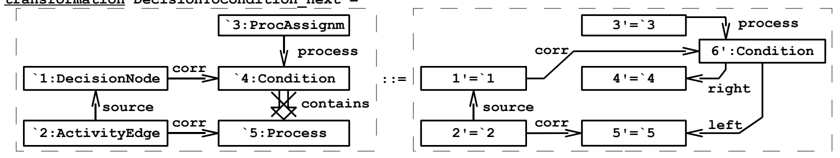
Concluding, we see the particular advantage of our solution in the straightforward and declarative way of specifying a transformation. We require no additional control structure nor priorities on rules. This greatly improves the maintainability and comprehensibility of the transformations. Furthermore, it is possible to bidirectionally interpret the relational rules.

4.3 Solution using PROGRES

The *Programmed Graph REwrite System* [24] is a general-purpose graph rewriting language. Its expressive graph language and the mature environment (including static analyzers, a debugger and code generator) encouraged its application in the tool contest. However, PROGRES does not explicitly support model transformation features like automated traceability management or bidirectionality.

The PROGRES-based solution comprises a single graph schema for both meta-models, plus a single interconnecting edge type to store traceability links. Model transformation rules are roughly structured as follows: The left-hand side (*LHS*) queries subgraphs of both (source and target) models connected by traceability edges, forming the transformation's *context*. In addition, an *increment* of the source model, which should be transformed by the current rule, is connected to the LHS's context. Non-recursive processing is guaranteed by negative application conditions (*NAC*s), which ensure that no element in the target model exists for the given source increment. On the right-hand side (*RHS*) of rules, a corresponding element is created in the target model and connected to the processed source increment.

```
transformation DecisionToCondition_next =
```



```
 transfer 6'.expr := `2.guard;
end;

path contains : BinaryOp -> ProcExp =
    ( ( -left-> or -right-> ) & instance of BinaryOp )* & ( -left-> or -right-> ) end;
```

Fig. 9. PROGRES transformation rule & textual path expression

Special handling is required for correspondences mapping n-ary source increments to binary target increments. As an example, the handling of DecisionNodes is split into two. First, an initial transformation rule maps their respective else-branch and an arbitrary other branch to a corresponding Condition. Afterwards, the rule depicted in Figure 9 is applied as long as possible to remove the Condition (`‘4`) of ProcessAssignment from its container and to add it as right child to a newly created Condition (`6’`). The termination of the transformation is guaranteed in this case by the negated path condition contains. This condition (depicted textually in the figure) ensures that the candidate Process `‘5` is not reachable from node `‘4` via left or right edges.

The PROGRES-based solution does not explicitly model control-flow, but relies on a non-deterministic rule application following an as-long-as-possible manner. Therefore, no dataflow passing “current” elements along with rule invocations is necessary. Termination is guaranteed by the guards discussed above, and by the fact that a traceability link is created by each rule application.

From the created specification, an executable prototype can be generated which is able to visually present UML activity diagrams and the resulting CSP expressions. Besides, GXL-based graph exchange and a textual output for CSP expressions is available. Activity diagrams can be edited using a set of consistency-preserving graph transformation rules.

4.4 Solution using GrGen.NET

The basic idea of our approach is to process the UML graph in a topological order. The working set is determined by specially marked edges (by type) and negative application conditions (NACs). During the transformation process each piece of the UML graph is removed as the according CSP graph elements are created.

As GrGen.NET provides all the necessary primitives, the UML and CSP meta models can be expressed directly (see [13]). Especially the `ActivityEdge` can be modeled by an edge type (as opposed to nodes in the given UML meta model) because the type system allows attributed edges.

Moreover, GrGen.NET provides basic support for the transformation of models to text (unparsing). However, more expressive support could alleviate the user from the overhead of specifying rules and control flow for unparsing.

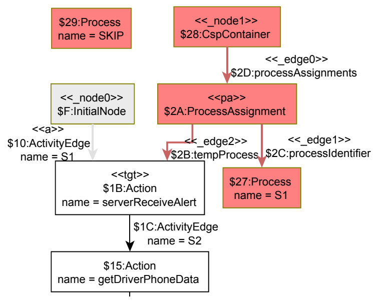


Fig. 10. The TFInitial rule applied to the example graph

Figure 10 shows a screenshot of the debugger of GrGen.NET during a rewrite step, which removes UML elements and builds up the according CSP elements. The red (dark grey) graph elements have just been created, whereas the light grey graph elements will be deleted. The names of the rule elements are given in angle brackets.

Using the standard settings of GrGen.NET, the transformation including the text output only takes about 100 ms. This even includes the overhead for just-in-time compilation, which accounts for about 99% of the execution time.

4.5 Solution using GReAT

GReAT [3] is a metamodel based transformation tool implemented within the framework of GME [19]. GReAT offers several features that make designing and implementing transformations intuitive and simple. The metamodels of the source and target languages are specified using UML class diagrams, with the additional capacity to define cross metamodel entities and temporary global objects which can be accessed in any rule of the transformation. A data-flow like model is used for sequencing transformation rules, added with conditional execution of rules (using a boolean *Guard* condition) and conditional branching. The GReAT solution for this case study illustrates the use of some of these options.

An interesting part of the UML to CSP case study was the transformation of `Decision` and `Fork` nodes. The challenge was to construct a binary tree structure from a list of arbitrary length, such that the last Condition node has two Process type children. The strategy adopted for transforming `Decision` nodes is: (1) When encountering a `Decision` node, take the first outgoing edge. Create a `Condition`, whose *lhs* is the associated `Process`, and the *rhs* is a new `Condition`; (2) For the next `Activity Edge`, create an *lhs* for the associated `Process` on the last `Condition`, and a new `Condition` as *rhs*. This is repeated for all the edges that are not marked “else” in the `Decision` node; (3) Finally, when only the edge marked “else” is left, the last remaining empty *rhs* `Condition` is replaced with a `Process` corresponding to the last edge. This requires collecting all the `Decision` nodes in the input model, and performing a sequence of operations for each `Decision` node. The layout of the transformation rules in GReAT is shown in Figure 11.

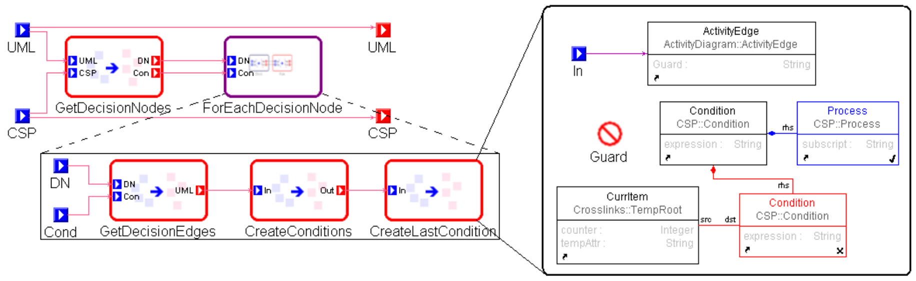


Fig. 11. Rule Sequencing and Rule Detail in GReAT

The rule *CreateConditions* is executed conditionally, for `Activity Edges` that do not have an "else" guard. This creates a binary tree of `Condition` nodes in the output, with each node having a `Process` as its *lhs* child, and another `Condition` node as its *rhs* child. When the "else" edge is encountered, the last *rhs* `Condition` node child is replaced with a `Process`. This is done by the rule *CreateLastCondition* as shown in Figure 11. *CurrItem* is a global object, which is used to track the last `Condition` node in the current binary tree. The rule *CreateLastCondition* deletes the last *rhs* `Condition` and creates a new `Process` in its place.

In addition to these features, GReAT comes with a code generator to generate more efficient transformations in C++, and an interactive debugger. A complete overview of the GReAT toolkit can be found in [7].

4.6 Solution using VMTS

In VMTS environment [1], we have created the metamodels of the activity diagrams and the CSP diagrams according to the specification of the case study. Metamodel based modeling and validation is supported: metamodels are used during the whole transformation process to describe models and to validate them in each transformation step.

The transformation is defined with a control flow (using the notation of the UML activity diagrams), which consists of separate transformation steps as depicted in Fig. 12.

Each transformation step is a graph rewriting rule defined with a left hand side and a right hand side graph. The transformation control flow describes the order of the transformation steps with directed edges between the nodes; it receives an input model (an instance of the activity diagram metamodel) and produces a newly created output model (an instance of the CSP metamodel). The most important properties of the transformation control flow are the following:

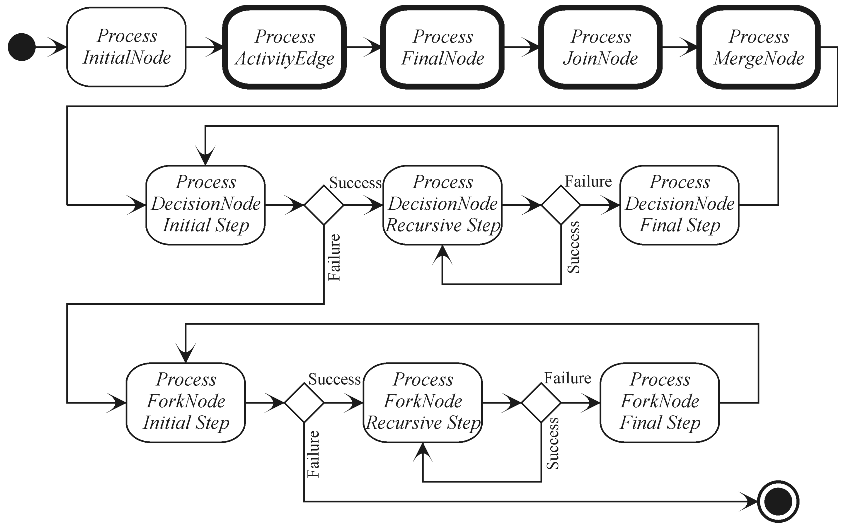

Process InitialNode
Process ActivityEdge
Process FinalNode
Process JoinNode
Process MergeNode
Process DecisionNode Initial Step
Success
Failure
Process DecisionNode Recursive Step
Failure
Success
Process DecisionNode Final Step
Process ForkNode Initial Step
Success
Failure
Process ForkNode Recursive Step
Failure
Success
Process ForkNode Final Step

Fig. 12. The control flow of the transformation in VMTS

1. Some rules are exhaustive rules, which means that before we proceed to the next rule, we apply the current rule repeatedly while the input model can be matched.
2. By changing the value of a special attribute (*IsProcessed*) owned by each element belonging to the input activity diagram, we guarantee to process each element at most once during the transformation rules, hereby the transformation process surely terminates.
3. Each branch node of the control flow is left by two edges, the processing flow follows one of them if the previous rule was successfully applied, or the other one if the previous rule could not be applied.
4. With internal causalities, it is possible to identify an element on the left hand side of a rule with an element on the right hand side of the same rule.
5. With external causalities (also known as parameter passing), we can identify an element on the right hand side of a rule with the element of the left hand side of the next rule in the control flow.

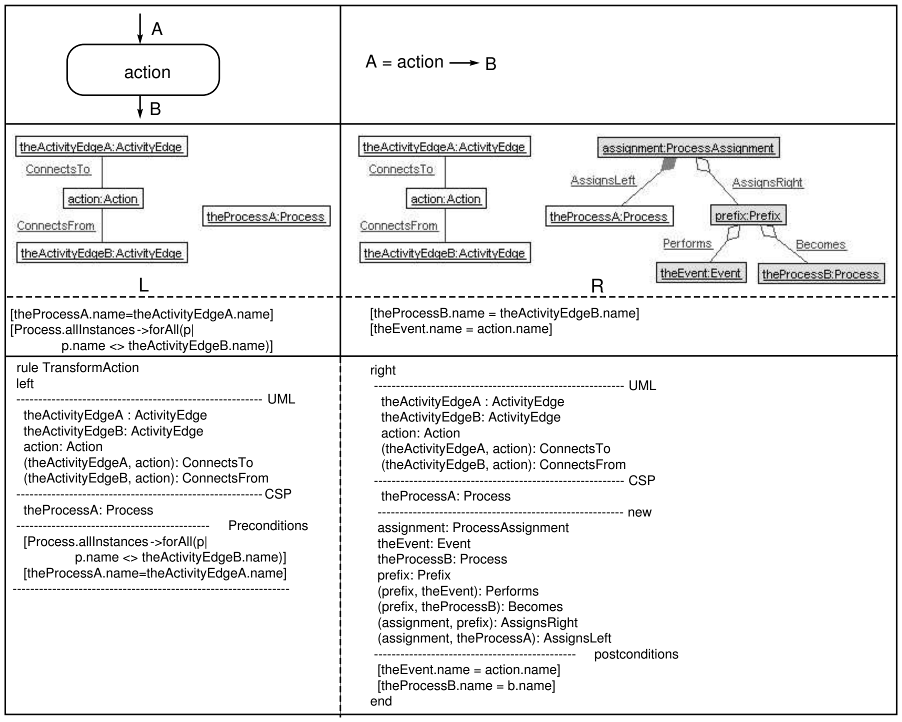
The Traversing Processor (TP) is part of VMTS tool. In the first step, it generates a C# API based on a chosen metamodel. Using TP we can execute the code by providing an instance model of the current metamodel as an input. By modifying the generated source code, any processing algorithm can be easily realized. In this case we use TP to produce the CSP expressions in a plain text format from a successfully created CSP model.

The result of the transformation is deterministic and the termination of the transformation process is guaranteed, because of the special attributes that ensure that each rule can be applied only finite number of times during the transformation.

4.7 Solution using USE

This section presents a solution with USE (UML-based Specification Environment) [14], which combines UML and OCL for specifying transformations.

For the case study, the metamodels of UML activity diagrams and CSP processes are directly expressed in USE as class diagrams attached with OCL invariants. The host graphs are presented as object diagrams.



```
[theProcessA.name=theActivityEdgeA.name]
[Process.allInstances->forAll(p|
          p.name <> theActivityEdgeB.name)]
```

```
[theProcessB.name = theActivityEdgeB.name]
[theEvent.name = action.name]
```

```
rule TransformAction
left
------------------------------------------------------- UML
  theActivityEdgeA : ActivityEdge
  theActivityEdgeB: ActivityEdge
  action: Action
  (theActivityEdgeA, action): ConnectsTo
  (theActivityEdgeB, action): ConnectsFrom
------------------------------------------------------- CSP
  theProcessA: Process
-------------------------------------------    Preconditions
  [Process.allInstances->forAll(p|
          p.name <> theActivityEdgeB.name)]
  [theProcessA.name=theActivityEdgeA.name]
---------------------------------------------------------------
```

```
right
------------------------------------------------------- UML
  theActivityEdgeA : ActivityEdge
  theActivityEdgeB: ActivityEdge
  action: Action
  (theActivityEdgeA, action): ConnectsTo
  (theActivityEdgeB, action): ConnectsFrom
------------------------------------------------------- CSP
  theProcessA: Process
  ----------------------------------------------------- new
  assignment: ProcessAssignment
  theEvent: Event
  theProcessB: Process
  prefix: Prefix
  (prefix, theEvent): Performs
  (prefix, theProcessB): Becomes
  (assignment, prefix): AssignsRight
  (assignment, theProcessA): AssignsLeft
  --------------------------------------------    postconditions
  [theEvent.name = action.name]
  [theProcessB.name = b.name]
end
```

Fig. 13. Realizing the rule TransformAction with USE

The figure 13 shows a formulation of the rule TransformAction with USE. Matching a rule is carried out by evaluating OCL queries on the source object diagram. These queries are captured by the precondition of the operation corresponding to the rule. In this case, we obtain objects for the nodes on the left-hand side as the input of the operation. Applying the rule by USE commands realizing the rule, we create objects and links for the right-hand side. After each rule application, one may check the postconditions of the rule for an on-the-fly verification of the transformation. The sequence of rule applications can be presented by a sequence diagram.

Rules in USE are captured in a dedicated language, which are then automatically translated into USE command sequences and OCL pre- and postconditions by the OCL generator of USE.

The example transformation is always checked after each rule application. By that, USE detects that the original metamodels from the case study had to be adjusted: some composition relationships had to be changed to aggregation relationships. Otherwise the object diagram representing the CSP process of the case study is not a well-formed instance of the CSP metamodel. In addition, our approach allows to integrate OCL invariants on the metamodels, which can be checked after each transformation step. For example, the following OCL invariant expressed that "the assignments have pairwise distinct left hand sides in a CSP container ":

```
context CspContainer
inv distinctProcessNames:
processAssignment->forAll(p1,p2| p1<>p2 implies p1.left.name <> p2.left.name)
```

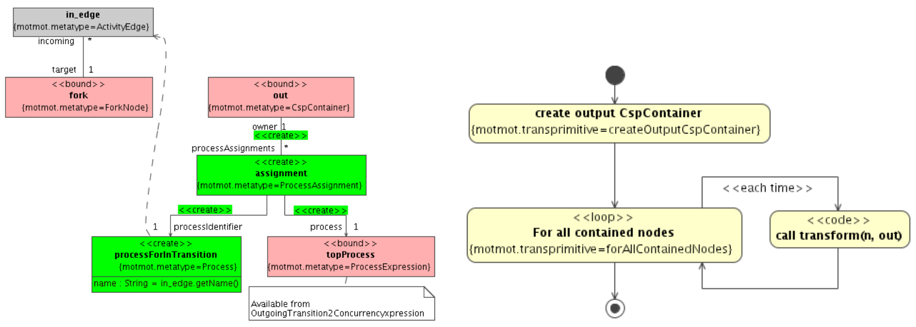
4.8 Solution using MoTMoT

MoTMoT is a tool that transforms UML models of controlled graph transformations into executable Java code that can access model repositories in a JMI or XMI standard compliant way. It has been designed to illustrate how several model transformation problems of the Fujaba tool can be solved.

MoTMoT enables one to specify primitive graph transformation rules (so-called *Story Patterns*) and control structures (so-called *Story Diagrams*) with any UML 1 standard-compliant modeling tool, instead of forcing transformation writers/maintainers to use the dedicated Fujaba editor. UML-to-CSP case study has been solved for mapping input activity diagrams from off-the-shelf UML 2 editors such as MagicDraw 10. Note that other submissions force the use of an ad-hoc (i.e., case-study specific) UML editor for producing input activity diagrams [27].

Figure 14 (a) presents an example rewrite rule in MoTMoT/Fujaba syntax. Remark that ≪*bound*≫ node variables either represent nodes that have already been bound by previously executed rules (e.g., *topProcess*), or nodes that are available as method parameters (e.g., *fork*, *out*). Node and edge variables marked with ≪*create*≫ are created by the rewrite rule. Finally, nodes and edges without such markers need to be matched in the host graph. With this semantics in mind, Figure 14 (a) shows the rule for mapping an input *Fork* node to an output CSP expression.

Figure 14 (b) shows how the MoTMoT transformation ensures that each input node is transformed exactly once: the Story Diagram models that after the creation of the output CSP container, the transformation should match each input node exactly once (using the iterative ≪*loop*≫ construct). For each match, a *transform(inputElement, outputContainer)* method is called. This method is implemented for each type of activity node and is modeled by diagrams such as Figure 14 (a).


in_edge
{motmot.metatype=ActivityEdge}
incoming
*
target
1
<<bound>>
fork
{motmot.metatype=ForkNode}
<<bound>>
out
{motmot.metatype=CspContainer}
owner
1
<<create>>
processAssignments
*
<<create>>
assignment
{motmot.metatype=ProcessAssignment}
<<create>>
<<create>>
1
processIdentifier
process
1
<<create>>
processForInTransition
{motmot.metatype=Process}
name : String = in_edge.getName()
<<bound>>
topProcess
{motmot.metatype=ProcessExpression}
Available from
OutgoingTransition2Concurrencyxpression
create output CspContainer
{motmot.transprimitive=createOutputCspContainer}
<<each time>>
<<loop>>
For all contained nodes
{motmot.transprimitive=forAllContainedNodes}
<<code>>
call transform(n, out)

(a) Out-place rewrite rule for Fork nodes

(b) Control flow: transform each input node

Fig. 14. Story Driven Modeling: Story Pattern and Story Diagram examples.

A first strength of the MoTMoT submission is the utilization of colors and layout patterns to improve the readability of a transformation model. Secondly, complex transformation rules are decomposed into manageable units by means of views on such rewrite rules. For example, Figure 14 (a) only shows the core mapping concerns for mapping a UML *Fork* node to a CSP *Process Assigment*.

More technical concerns are modeled by another view on the same rewrite rule. A third strength of MoTMoT is its conformance with OMG's MDA standards. For the UML-to-CSP case study specifically, we have illustrated how input from non-standard tools can be consumed as well, using in-place transformation rules. Finally, the submission illustrates MoTMoT's extensibility by applying language constructs that are realized by means of higher-order transformations.

The first drawback is the limited "out-of-the-box" usability: when only specializing a generic UML editor with the UML profile for Story Diagrams, some domain-specific editor features (such as advanced auto-completion) are not available. In practice, one may therefore want to extend one's favorite UML tool with a (small) plugin for such features. As a second drawback, the submission illustrates that some platform specific details cannot (yet) be hidden from a MoTMoT transformation model.

4.9 Solution using GrTP

The aim of this case study is to build a model transformation which takes a UML activity diagram as an input and gives a list of CSP processes as an output. The initial activity diagram can be produced by means of transformation-based Graphical Tool-building Platform called GrTP [8]. The platform (regardless of other facilities) allows users to make an activity diagram based on the UML activity diagram meta-model and execute the transformation called *UMLtoCSP* which transforms the UML model to a CSP model and then verifies it upon termination and determinism up to process equivalence.

The result of the transformation consists of several parts:

1. a CSP model - an instance of the CSP meta-model;
2. a text file containing the list of CSP processes together with their assignments according to the textual syntax of CSP;
3. an answer to the question "Does the CSP model execution terminate?";
4. an answer to the question "Is the CSP model deterministic?".

For the GrTP platform to be able to work efficiently, a novel model transformation language L0 [2] has been implemented with a highly efficient compiler. The transformation *UMLtoCSP* is also written in the language L0. The language L0 is very simple and completely procedural, and it has only a textual syntax.

The solution of the problem is mainly based on rules given in Section 2. In addition, a rule containing information about conditions without an else branch is added (Figure 15). The termination is verified partly - it is only possible to assure the CSP model execution terminates if it contains no cycles. The CSP determinism in this case is defined in this manner - the CSP model is deterministic if and only the following condition holds for each process expression starting from the initial process:

1. every symbol of the given alphabet leads to at most one process from a given state;
2. if some symbol of the alphabet leads to more than one process, then bisimulation holds between such processes.

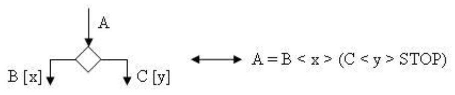


Fig. 15. Extra rule - a condition without an else branch.

The advantage of the tool used to solve the task is in its simplicity from the view point of an end user: no installation is required, no complicated instructions need to be learned, — although, obviously, a new language has to be learned. However, since the tool allows user to make arbitrary UML activity diagrams, the result of the *UMLtoCSP* transformation cannot be predicted in some (erroneous) cases.

4.10 Solution using MOMENT2-GT

MOMENT2-GT [5] is a graph transformation tool where graphs are provided as MOF-based models and production rules are defined in a QVT-based textual format. In MOMENT2-GT, a graph transformation definition is compiled into a rewrite theory in Maude [11], the input graph is represented as a term of a specific sort that is defined in this theory, and the execution of a graph transformation is handled by Maude's algorithm for term rewriting modulo associativity and commutativity. Graph transformations are performed by following the *Single Pushout* approach.

In our solution, we process the objects that constitute the input activity model generating objects in the resulting CSP model. The idea behind the transformation definition is to delete activity nodes whenever they have been processed. We have studied two solutions for the case study by taking into account dangling edges implicitly or explicitly. In the first case, MOMENT2-GT takes care of possibly generated dangling edges. In the second case, the user must avoid their generation in the transformation definition. Both solutions can be downloaded from [5].

We provide an average of the time measurements that have been obtained during 10 experiments[11]. The transformation that can produce dangling edges was performed in an average time of 1431.2 ms by Maude. The transformation that was designed to avoid dangling edges was performed in an average time of 885.4 ms by Maude.

MOMENT2-GT is based on a Maude algebraic specification of Essential MOF that is provided as a plugin to EMF. This means that EMF models can be directly used as formal entities in the algebraic framework, where they can be treated as graphs or as terms. Therefore, we can apply Maude-based formal analysis techniques [11], such as model checking of invariants or LTL model checking, to model-based systems in a straightforward way.

A disadvantage in our approach is that it lacks of graphical concrete syntax. Comparing a production rule in Tiger and in MOMENT2-GT (as illustrated in Fig. 16) shows, at a first glance, that our approach is not the most appropriate for communication purposes. However, for expert users, a textual-based syntax may offer editing facilities that are difficult to achieve in a graphical approach: copy & paste, text replacement, etc. In addition, MOMENT2-GT constitutes a framework that is defined at a high level of abstraction in Maude. Therefore it is ideal for experimenting with new model transformation features, keeping in mind a realistic approach in terms of efficiency.

4.11 Solution using XL

The case study can be implemented easily using the textual programming language XL on the basis of the graph of GroIMP [18]. At first, we have to translate the meta models to a Java class hierarchy which can be done as part of the XL code as in

```
abstract module ActivityNode extends Node;
```

[11] The experiments have been performed on a Core DUO 2Ghz with 2Gb RAM, using Ubuntu 7.04.

```
eq 'rule2-ActivityEdge-->ProcessIdentifier
{
    lhs 'source {
        '1 : 'ActivityEdge {
            'name = 'inEdgeName:String
        }
        '2 : 'CspContainer { }
    };


    rhs 'source {
        '1 : 'ActivityEdge {
            'name = 'inEdgeName:String
        }
        '2 : 'CspContainer {
            'processAssignments = 'pa : 'ProcessAssignment {
                'processIdentifier = 'sp : 'ProcessIdentifier {
                    'name = 'inEdgeName:String
                }
            }
        }
    };

    nac 'source 'noProcess {
        'sp : 'ProcessIdentifier {
            'name = 'inEdgeName:String
        }
    } ;
}
```

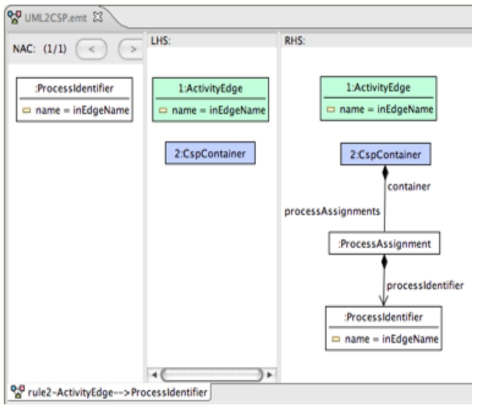


Fig. 16. Production rule in MOMENT2-GT and Tiger.

Secondly, we have to instantiate the meta model with the source UML graph. Ideally, we would use a graphical editor or some common graph exchange format which can be imported into our system. Unfortunately, this is not yet possible so that we have to input the source graph as part of the XL code, too:

```
Axiom ==>> ^ InitialNode
           -ActivityEdge("S1")-> Action("serverReceiveAlert")  ...;
```

Thirdly, we have to specify the transformation rules and to control their application. For the rules we make use of the fact that UML activity edges play the role of CSP processes and UML actions play the role of CSP events. Thus, we can keep these nodes in the graph as context for the gluing of the SPO approach, perform all transformations as if these nodes were both UML and CSP nodes, and replace them with their actual CSP nodes as a final step. As an example, the rule for an action

```
a:ActivityEdge -o-> x:Action -i-> b:ActivityEdge ==>>
    ^ -processAssignments-> ProcessAssignment [-identifier-> a]
      -process-> Prefix [-event-> x] -targetProcess-> b;
```

already creates some CSP nodes and edges, but keeps the UML nodes of the left-hand side. Only after the final step we have a valid CSP graph:

```
a:ActivityEdge ==>> p:Process(a.getName()) moveIncoming(a, p, -1);
a:Action ==>> e:Event(a.getName()) moveIncoming(a, e, -1);
```

Concerning the control of rule application, we make use of XL extending Java: a rule is executed simply when it is reached (as a statement) by the usual control flow of Java. Furthermore, we may set the mode of rule application, either parallel or sequential. It turns out that most rules can be applied in parallel in an initial step with the exception of the creation of binary expression trees for UML decision and fork nodes which has to be done sequentially afterwards.

Among the three case studies of AGTIVE 2007, the UML-to-CSP case study was least related to the principal application domain of XL. Nevertheless, it was easily possible to implement the transformation. However, our system does not provide means for verification.

5 Lessons Learned

According to the categories discussed in Sec. 3, we can draw the following conclusions, which are summarized also in Table 1.

- **Modeling (metamodeling) framework**: Each tool offers an underlying model manipulation and metamodeling framework to support transformations. Supported features frequently included standards-compliant metamodels (like EMF or GXL) well-formedness constraints for a modeling language (typically expressed in OCL), edge attributes, etc. Several solutions used UML diagrams for capturing metamodels and model. Furthermore, some transformations were built above a full-fledged domain-specific modeling framework.
- **Rule language**. Solutions used either a textual or a graphical language for capturing transformation rules. Some tools integrated relied on standards-compliant languages in transformation design such as OCL (as in case of USE or VMTS) or QVT (in case of MOMENT2-GT). Interestingly, none of the tools provided *both* a graphical and a textual language for capturing rules.
- **Control structures**. Control structures used in at least one of the solutions included parameter passing (e.g. GReAT and VMTS), parallel rule execution (e.g. XL), as long as possible rule application, topological (hierarchical) ordering enforced by rules and rewrite sequences (in GRGEN.NET), a dataflow-based language (in GReAT), and traditional programming constructs like conditional branching or loops. XL used native Java constructs as control structures for the transformation. The TIGER, the TGG and the MOMENT2-GT solutions were purely declarative, i.e., they did not use any control structure.
- **Handling each match once**. In order to process each match only once, different solutions used either some explicit helper data structure (such as a reference model or a helper attribute), negative application conditions for rules, and control structures like *foreach*. The TGG solution automatically maintains all instances of the applied rules to remember the matched nodes. Finally, some solutions (such as GRGEN.NET, GReAT or XL) removed some (or all) elements of the source model one by one to prevent multiple application of rules on the same match.
- **Advanced GUI features**. Advanced features of the graphical user interface of different tools included graphical editors (e.g. in case of TIGER, TGGs

Solution	Metamodeling	Rule language	Control structure (in the solution)	Handling each match once	Analysis support (in the tool)	Advanced transf. features
Tiger	EMF	graphical GT rules	none	reference metamodel & NAC	–	compiled GT rules
TGG	EMF	TGGs (graphical)	none	implicitly handled by TGG semantics	–	bidirectionality
PROGRES	graph schema	visual GT rules	non-det. rule appl., as long as possible	NAC	graph constraints	path expressions, backtracking
GrGen.NET	custom domain-specific	textual GT rules	sequential composition of rules	deconstruct the source graph	connection assertions, interactive debugger	transactions
GReAT	UML Class Diagrams	Graphical, UML-like notation with Boolean Guards	Explicit sequencing of rules with Dataflow like syntax	Implicit	Interactive Debugger	Closure over matches to form groups
VMTS	domain independent, n-level metamodeling framework	graphical rules with OCL	activity diagram with parameter passing	helper attributes and OCL constraints	run-time validation of OCL constraints	explicit traceability
USE	MOF and EMF can be explicitly modeled	textual GT rules	scripting and redex computation	with NACs	checking of (pre- and post) conditions and invariants	bidirectionality possible
MoTMoT	standards compliant (MOF, UML, JMI)	graphical GT rules	Story diagrams	helper structures	–	higher-order transformations
GrTP	UML	textual model transformation language L0	textual structures (foreach etc.)	using foreach construct	verif. of termination and determinism (of the target model)	–
MOMENT2-GT	EMF (compiled to Maude)	textual QVT-based GT rules	none	NAC, OCL constraints	formal analysis provided by Maude	–
XL	Java class hierarchy	textual, extends Java	parallel, seq., as long as possible, Java	removal of used UML nodes	–	–

Table 1. Comparison of solutions and tools

or GrTP), different views of rules (e.g. in MoTMoT), and editors of the source and target models (e.g., in PROGRES, VMTS, GrTP). Online and interactive layout of the host graph is present in GrGen.NET.
- **Underlying run-time transformation platform**. Most of the tools were implemented in Java, several of them above industrial modeling platforms like EMF (in case of e.g TIGER or TGG) or JMI (in case of MoTMoT). The exceptions include VMTS and GrGen.NET, which used .Net as underlying platform. PROGRES transformations can be compiled into C, GReAT and GrTP transformations can be compiled into C++. Finally, MOMENT2-GT transformations are executed within the Maude rewriting framework.
- **Analysis support.** Some tools provided support for analyzing the models or the transformations. OCL-based validation of models were reported in USE and VMTS, where the latter also supports the run-time validation of constraints during transformation. An interactive debugger is available in GReAT and GrGen.NET. Formal analysis of transformation specifications is available in MOMENT2-GT as provided by the underlying Maude engine.
- **Advanced model transformation features.** Some advanced model transformation constructs have also been used in different solutions. The TGG solution was the only solution supporting the bidirectionality of transformations. Higher-order transformations were used in MoTMoT.

As a concluding remark, let us identify some areas where existing tool support is not as extensive. Interestingly, none of the tools supported *implicit traceability* when all correspondence structures are derived automatically when applying the transformation rules. Such a solution is present in model transformation frameworks like ATL [6]. Instead, all solutions used some kind of explicit traceability (i.e. manually introduced correspondence structure) information to represent the interconnection of source and target models. Furthermore, incremental transformations were not supported by any of the tools, which is also a key issue in the design of model transformations. Existing analysis support available in the presented tools can only guarantee some correctness criteria for specific runs of a transformation, while there is a lack of support for *reasoning on the transformation (rule) level*. Finally, solutions did not emphasize the *reusability support* available in the corresponding tools, which is a critical aspect when developing complex transformations.

References

1. Visual Modelling and Transformation System (VMTS). `http://vmts.aut.bme.hu`.
2. The Lx transformation language set, 2007. `http://Lx.mii.lu.lv`.
3. A. Agrawal, G. Karsai, and A. Ledeczi. An end-to-end domain-driven software development framework. In *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Anaheim, California, 2003.
4. C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink

and J. Warmer (eds.), *Model Driven Architecture - Foundations and Applications: Second European Conference*, vol. 4066 of *LNCS*, pp. 361–375. Springer Verlag, Heidelberg, 2006.

5. Artur Boronat. The MOMENT2-GT web site, 2008. `http://www.cs.le.ac.uk/people/aboronat/tools/moment2-gt`.
6. ATLAS Group. *The ATLAS Transformation Language*. Available from `http://www.eclipse.org/gmt`.
7. D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: GReAT. In *3rd International Workshop on Graph Based Tools (GraBaTs 2006)*. Natal (Brazil), 2006.
8. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, and A. Sprogis. GrTP: Transformation based graphical tool building platform. In *MDDAUI 2007: Workshop on Model Driven Development of Advanced User Interfaces (Satellite event of MODELS 2007)*. 2007.
9. D. Bisztray. Verification of architectural refactoring rules. Tech. rep., Department of Computer Science, University of Leicester, 2008. `http://www.cs.le.ac.uk/people/dab24/refactoring-techrep.pdf`.
10. D. Bisztray and R. Heckel. Rule-level verification of business process transformations using CSP. In *Proc of 6th International Workshop on Graph Transformations and Visual Modeling Techniques (GTVMT'07)*. 2007.
11. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*. Springer LNCS Vol. 4350, 2007.
12. Eclipse Consortium. *Eclipse Modeling Framework (EMF) – Version 2.3*, 2007. `http://www.eclipse.org/emf`.
13. R. Geiß and M. Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In this volume.
14. M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, vol. 69:pp. 27–34, 2007.
15. J. Greenyer and E. Kindler. Reconciling TGGs with QVT. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil (eds.), *Proc. 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007, Nashville, USA*, vol. 4735 of *LNCS*, pp. 16–30. Springer, 2007.
16. J. Greenyer, E. Kindler, J. Rieke, and O. Travkin. TGGs for Transforming UML to CSP: Contribution to the ACTIVE 2007 Graph Transformation Tools Contest. Tech. Rep. tr-ri-08-287, Software Engineering Group, Dept. of Computer Science, Univ. of Paderborn, 2008. `http://www.uni-paderborn.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2008/tr-ri-08-287.pdf`.
17. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, 1985.
18. O. Kniemeyer and W. Kurth. The modelling platform GroIMP and the programming language XL. In this volume.
19. A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, vol. 34(11):pp. 44–51, 2001.
20. Object Management Group (OMG). MOF QVT Final Adopted Specification, 2007. `http://www.omg.org/cgi-bin/apps/doc?ptc/07-07-07.pdf`.
21. OMG. *Unified Modeling Language, version 2.1.1*, 2006. Http://www.omg.org/technology/documents/formal/uml.htm.
22. A. Rensink and G. Taentzer. AGTIVE 2007 Graph Transformation Tool Contest. In this volume.

23. A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer (ed.), *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, vol. 903 of *Lecture Notes in Computer Science (LNCS)*, pp. 151–163. Springer Verlag, Heidelberg, 1994.
24. A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.), *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, pp. 487–550. Volume 2. World Scientific, 1999.
25. *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*. `http://www.sensoria-ist.eu`.
26. Tiger Developer Team. *Tiger EMF Transformer*, 2007. URL `http://www.tfs.cs.tu-berlin.de/emftrans`. `http://www.tfs.cs.tu-berlin.de/emftrans`.
27. P. Van Gorp, O. Muliawan, A. Keller, and D. Janssens. Executing a platform independent model of the UML-to-CSP transformation on a commercial platform. In *AGTIVE 2007 Tool Contest*. 2007. `http://gtcases.cs.utwente.nl/wiki/UMLToCSP/MoTMoT`.
28. R. Wagner. Developing Model Transformations with Fujaba. In H. Giese and B. Westfechtel (eds.), *Proc. 4^{th} International Fujaba Days 2006, Bayreuth, Germany*, vol. tr-ri-06-275 of *Techn. Rep.*, pp. 79–82. Univ. of Paderborn, 2006.