Using PROGRES for Transforming UML Activity Diagrams into CSP Expressions

Erhard Weinell and Ulrike Ranger

RWTH Aachen University of Technology
Department of Computer Science 3 (Software Engineering)
Ahornstraße 55, D-52074 Aachen, Germany
`Weinell|Ranger@cs.rwth-aachen.de`
http://se.rwth-aachen.de

1 Introduction

In this paper, we describe how an UML activity diagram can be transformed into a corresponding CSP expression by using the graph rewriting language PROGRES [1]. PROGRES allows to model the transformation process in a visual and declarative way, leading to a clear and easy to understand specification. PROGRES does not only offer an extensive modeling language, but also provides a sophisticated environment including a syntax-directed editor and a code generation facility. Thus, source code can be automatically generated from the modeled transformation process. Additionally, we execute this source code as *visual* prototype [2]. The prototype enables users to draw activity diagrams and to derive corresponding CSP expressions.

The transformation process followed in this paper is based on [3, 4]. However, we chose not to use the triple graph grammar approach (TGG) and thus do not create a correspondence graph relating the elements of the activity diagram and of the CSP expression. Instead, we directly connect corresponding elements by edges. We favored this approach for the following reasons: First, the correspondences proposed in [3, 4] are always of cardinality 1-to-1. So, an explicit correspondence graph is not needed. By directly connecting the elements by edges, the specification is significantly simplified and the runtime graph is less space consuming. Second, the translation process can be completely specified within PROGRES allowing to use the whole existing environment. Thus, a visual prototype can be easily generated for the specification.

The paper is structured as follows: Section 2 describes the PROGRES specification modeled for the transformation process. Afterwards, we show how a visual prototype can be generated from the PROGRES specification in Section 3. We conclude with a summary in Section 4.

2 Specification

This section briefly describes the PROGRES specification which we have modeled to implement the transformation process. Subsection 2.1 focuses on the graph schema defining the structure of UML activity diagrams and of CSP expressions. The graph transformation rules for translating an activity diagram into a CSP expression are shown in Subsection 2.2. PROGRES turns out to be well suited for modeling the transformation process as both the structure and the transformation rules demanded in [4] can be specified straightforward.

2.1 Graph Schema

The metamodels of UML activity diagrams and of CSP expressions can be easily specified within a PROGRES graph schema: For every metamodel class, a node type in the graph schema is defined covering the name of the metamodel class and the necessary attributes. The metamodel associations are modeled by edge types in the graph schema relating the node types according to the metamodels.

Besides the equivalents for the metamodel elements in the graph schema, only two more types have to be defined in the PROGRES graph schema: First, the edge type corr is used for modeling the correspondences between elements of an activity diagram and elements of an CSP expression. This type has a 1-to-1 cardinality as only one element of the activity diagram may be related to one element of the CSP expression and vice versa. Second, the node type ConcurrencyCLS has to be declared for realizing the multiple inheritance of class Concurrency of CSP metamodel.

For lack of space, we do not show the graph schema in this paper, but refer interested readers to [2] (including the PROGRES specification).

2.2 Graph Transformation Rules

Generally, a PROGRES graph transformation rule is composed of two graph patterns: The left-hand side pattern (LHS) describes the structure of a sub graph before applying the rule. The right-hand side pattern (RHS) determines the modification of the sub graph. Besides the two patterns, a PROGRES rule may cover attribute conditions and attribute modifications, pre- and postconditions, embedding rules, and returning of parameters. A detailed description of the language constructs is out of scope of this paper, but can be found in [1].

For the transformation process, the PROGRES specification covers one graph transformation rule for every transformation rule demanded in [4]. Besides these rules, only two more transformation rules are needed in the specification for initializing the transformation process. As the modeled PROGRES rules are straightforward, we only describe one sample transformation rule.

The rule in Figure 1 resembles the *action rule* described in [3]. The rule creates for an action ('2) in the activity diagram a corresponding event (10') in the CSP expression. The negative node '6 of type CspElement on the LHS ensures that the action '2 has not already been mapped onto an event in the CSP expression. For the action '2, the RHS causes the creation of an event 10' and further nodes and edges needed in the CSP expression. To store the relation between the action and the newly created event, an edge of type corr is inserted.

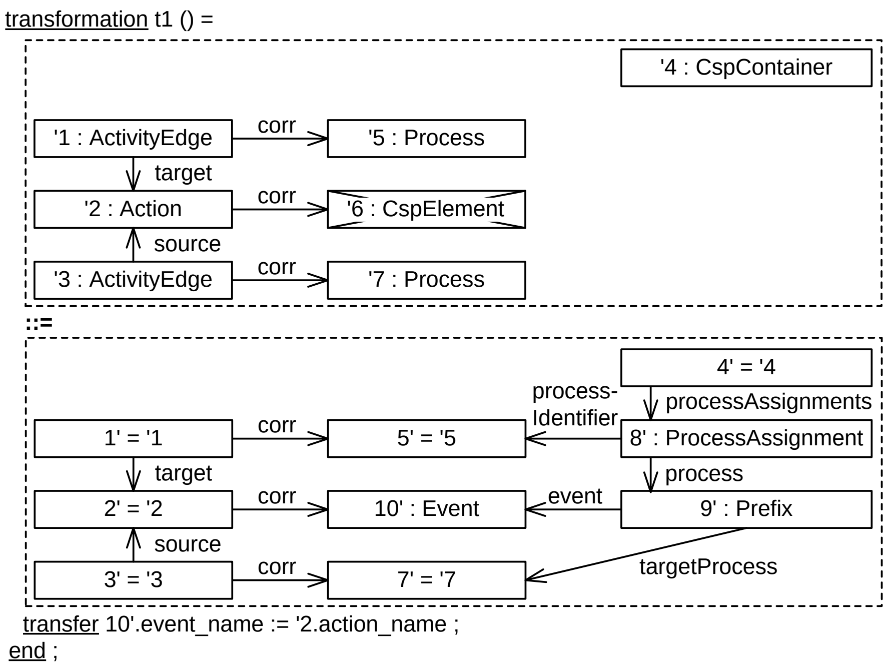


Fig. 1. Graph transformation rule t1

In addition to the demanded rules in [4], we have added rules for simplifying the finished CSP expression. Figure 2 shows one of these rules which removes trivial assignments, i. e. processes in the CSP expression consisting of only one process. To preserve the semantics of the CSP expression, we redirect all edges incident to the regarded process to the corresponding process expression.

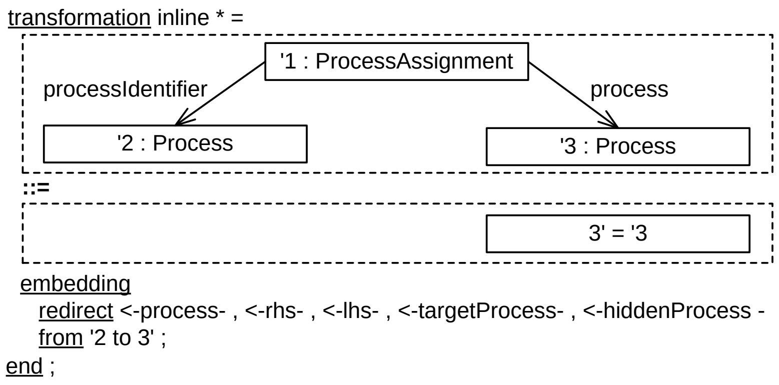


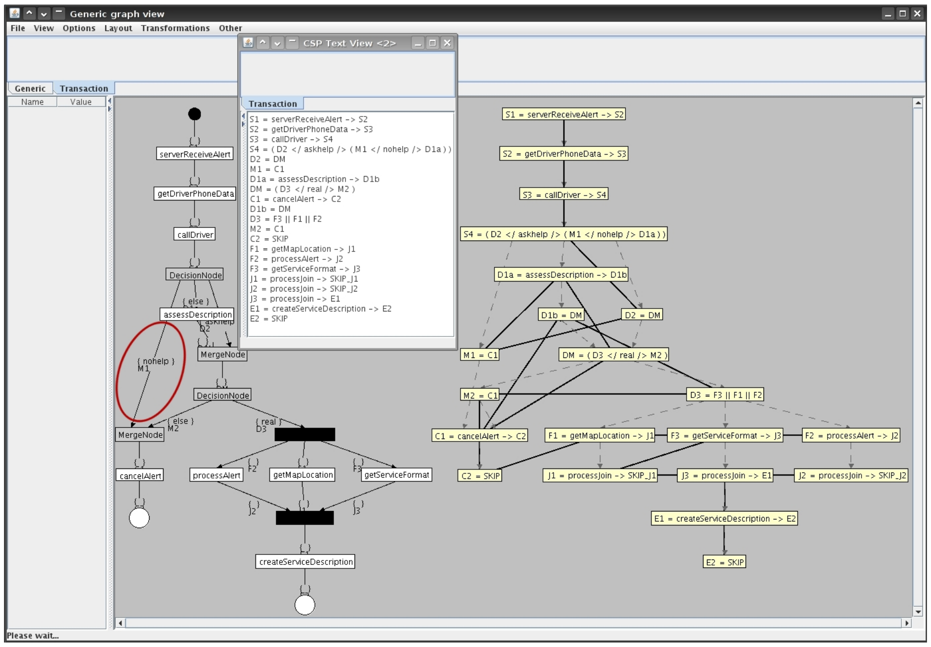
Fig. 2. Graph transformation rule inline

3 Prototype

From the PROGRES specification, we generated Java source code to build a prototypical diagram editor. This editor is able to display diagrams defined by the two metamodels and allows to call the transformation rules modeled in the PROGRES specification. The UPGRADE [5] framework was applied for visualization, so we only had to configure the prototype's view for displaying the diagrams.

Figure 3 shows the prototype after transforming the example diagram introduced in [4]. The figure displays the graphical representation of the activity diagram (left side) and the CSP expression (right side). In addition, a small window shows a textual representation of the CSP expression.

Using UPGRADE's configurable filter mechanism, activity diagrams and CSP expressions can be displayed in a user-friendly way. Initial and final nodes are depicted by circles, fork and join nodes by black bars. For actions, a white box with the name attribute's value is shown. In addition, UPGRADE allows to create edge-node-edge (ENE) filters, which display a node connected to two other nodes as an edge. According to the given UML metamodel, edges between activities are modeled as nodes of type ActivityEdge. An ENE filter displays these nodes as the edges shown in Figure 3. For example, the edge M1 shown in the figure is internally modeled as node with incident edges connecting to assessDescription and the MergeNode.



```
S1 = serverReceiveAlert -> S2
S2 = getDriverPhoneData -> S3
S3 = callDriver -> S4
S4 = ( D2 </ askhelp /> ( M1 </ nohelp /> D1a ) )
D2 = DM
M1 = C1
D1a = assessDescription -> D1b
DM = ( D3 </ real /> M2 )
C1 = cancelAlert -> C2
D1b = DM
D3 = F3 || F1 || F2
M2 = C1
C2 = SKIP
F1 = getMapLocation -> J1
F2 = processAlert -> J2
F3 = getServiceFormat -> J3
J1 = processJoin -> SKIP_J1
J2 = processJoin -> SKIP_J2
J3 = processJoin -> E1
E1 = createServiceDescription -> E2
E2 = SKIP
```

Fig. 3. Graphical & textual view of the prototype

```
S1 = serverReceiveAlert -> S2
S2 = getDriverPhoneData -> S3
S3 = callDriver -> S4
S4 = ( D2 </askhelp/> ( M1 </nohelp/> D1a ) )
D2 = DM
M1 = C1
D1a = assessDescription -> D1b
DM = ( D3 </real/> M2 )
C1 = cancelAlert -> C2
D1b = DM
D3 = F3 || F2 || F1
M2 = C1
C2 = SKIP
F3 = getServiceFormat -> J3
F2 = processAlert -> J2
F1 = getMapLocation -> J1
J3 = processJoin -> SKIP_J3
J2 = processJoin -> SKIP_J2
J1 = processJoin -> E1
E1 = createServiceDescription -> E2
E2 = SKIP
```

(a) Regular CSP Expression

```
S1 = serverReceiveAlert -> S2
S2 = getDriverPhoneData -> S3
S3 = callDriver -> S4
S4 = ( DM </askhelp/> ( C1 </nohelp/> D1a ) )
DM = ( D3 </real/> C1 )
D1a = assessDescription -> DM
C1 = cancelAlert -> SKIP
D3 = F3 || F1 || F2
F1 = getMapLocation -> J1
F2 = processAlert -> J2
F3 = getServiceFormat -> J3
J1 = processJoin -> SKIP_J1
J2 = processJoin -> SKIP_J2
J3 = processJoin -> E1
E1 = createServiceDescription -> SKIP
```

(b) After inline transformation

Fig. 4. Output after transformation to CSP

Although CSP expressions are stored as complex graph structures according to the CSP metamodel, the filter mechanism is able to create a convenient representation. Only nodes of type ProcessAssignment are displayed, using a derived attribute to obtain the depicted labels. Instances of all other types of the CSP metamodel and their interconnections are hidden from the user. Dashed lines between assignments represent static paths (derived edges) denoting a use relation. Informally, a use relation exists if an assignment's left side is referenced in another assignments right side. To give a textual representation of a CSP expression, we need to visit the ProcessAssignments in some order. This order is indicated by thick lines in the figure, where a breadth-first traversal was used.

The prototype allows users to create new UML activity diagrams and to modify existing ones. This editing functionality is also implemented by graph transformation rules. These rules ensure a basic set of well-formedness constraints, e.g. that only one Initial node exists. Rules may also be invoked using Python scripts and the integrated scripting host to enable batch processing.

For an easy validation of our results, Figure 4 depicts our tool's output after transforming the example diagram into a CSP expression. The output of the model transformation process is shown in Figure 4(a), whereas Figure 4(b) presents the state after removing trivial assignments by invoking the inline transformation rule. For example, occurrences of D2 were replaced by DM and the assignment D2 = DM was removed.

4 Conclusion

The PROGRES language showed to be well suited to capture the problem [4]. Both metamodels were implemented as a graph schema with very little modifications. The transformation process was modeled using graph transformation rules being executed as long as possible. NACs are used to detect entities transformed before. For this, a total of 14 transformation rules are used, which could be implemented straightforward according to [4]. Another 16 rules allow users to build activity diagrams and assist in the post-processing phase.

The complete project took approximately 16 hours for a trained person to write the transformation rules, add test data and build the basic prototype. This does not include time required for bug fixing, although PROGRES features numerous static checks and a build-in debugger.

The current prototype allows to build and transform UML activity diagrams. The textual output generated for CSP expressions can be used by other tools for further analysis. In addition, an import/export mechanism based on GXL is available to support integration with other tools. The first option might be the more convenient choice for existing tools designed for CSP analysis. The second one is based directly on graph structures, so further processing does not require to parse the output.

References

1. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: Language and environment. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. Vol. 2. 1st edn. World Scientific, Singapore (1999) 487–550
2. Weinell, E., Ranger, U.: Specification and visual prototype for transforming UML activity diagrams into CSP expressions (2007) http://agtive:toolcontest@se.rwth-aachen.de/files/agtive2007-toolcontest/UML_to_CSP.zip.
3. Bisztray, D., Heckel, R.: Rule-level verification of Business Process Transformations using CSP. In Ehrig, K., Giese, H., eds.: Graph Transformation and Visual Modeling Techniques, GTVMT'07. Vol. 6 of Electronic Communications of the EASST., European Association of Software Science and Technology (2007) 3–15
4. Bisztray, D., Ehrig, K., Heckel, R.: Case study: UML to CSP transformation (2007) Case Study of the AGTIVE 2007 Tool Contest, http://www.informatik.uni-marburg.de/ swt/agtive-contest/UML-to-CSP.pdf.
5. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: A framework for building graph-based interactive tools. In Mens, T., Schürr, A., Taentzer, G., eds.: 1st International Workshop on Graph-Based Tools, GraBaTs'02. Vol. 72 of Electronical Notes in Theoretical Computer Science., Elsevier Spektrum Akademischer Verlag (2002)

A Getting & using the prototype

This section gives basic information on how to get and run the prototypical editor. It is intended as reference for the reviewers, but is not part of the paper contribution. In case of having trouble with the prototype, please contact the authors.

A.1 Getting

– Our prototype is available for download at (case sensitive): http://se.rwth-aachen.de/files/agtive2007-toolcontest/UML_to_CSP.zip
  The filesize is approximately 14 megabytes.
– Unfortunately, due to the use of commercial libraries in the visualization framework, we cannot release this tool into the public domain. Therefore, the download is password-protected to limit the audience. Username:`agtive` Password:`toolcontest`

A.2 Starting the prototype

– The downloaded archive should be extracted somewhere.
– Make sure a Java SE 5 or 6 runtime environment is installed, and the `JAVA_HOME` environment variable is set.
– The prototype is started using `UML_to_CSP` (Linux) or `UML_to_CSP.bat` (Win32).
– The prototype was tested on recent Windows and Linux versions, we cannot guarantee its functionality on other platforms.

A.3 Running the example

– The demo transformation can be generated using Transformations - Examples - Contest Example.
– The generated diagram can be layouted using Layout - Hierarchical.
– Generating the according CSP expression is done by Transformations - Transform UML to CSP.
– Transformations - Lineralize Statements creates a linear order of process assignments, required for textual output.
– The text view can be opened with File - New - CSP Text View.
– The prototype should be closed via File - Exit.