

Case Study: UML to CSP Transformation

Dénes Bisztray, Karsten Ehrig, and Reiko Heckel

Department of Computer Science,
University of Leicester, United Kingdom
{dab24,karsten,reiko}@mcs.le.ac.uk

Keywords: graph transformation, model transformation, CSP, UML

1 Introduction

The aim of this case study is to introduce a transformation from activity diagrams [1] to Communicating Sequential Processes [2].

As the de-facto standard for software design, the UML [1] activity diagram is used to describe low level behaviour of software components or to represent workflow aspect of business processes. In both cases, the verification of the behaviour can be important. The purpose of verification can run from a simple liveness or termination check to the verification of refinement between redesigned model instances. To verify any aspect of behaviour, 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 processes by means of graph transformation.

The paper is organised as follows. In Section 2 we introduce the metamodels used for the source and target of the transformation. The details of the transformation method is presented in Section 3. In Section 4 the possible challenges are introduced for the tool environment. And finally in Section 5 an example transformation is presented.

2 Metamodels

In this section we introduce the source and target metamodels.

A source is the simplified metamodel for activity diagram based on [1]. It is shown in Figure 1.

The metamodel for CSP, as far as required for the case study, is shown in Figure 2. A *Process* is the behaviour 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.

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

Following the Composite Pattern [3], the abstract class *ProcessExpression* represents a guarded expression. It can be either a simple *Process P*, a *Prefix*

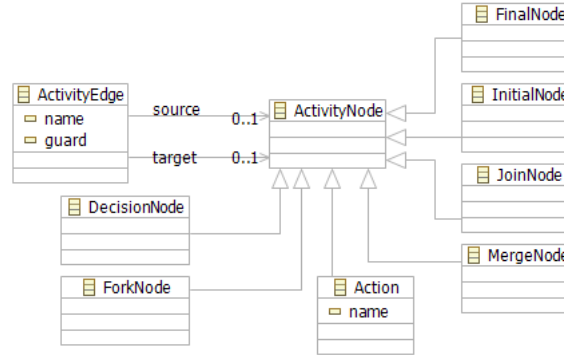


Fig. 1. Activity Diagram Metamodel

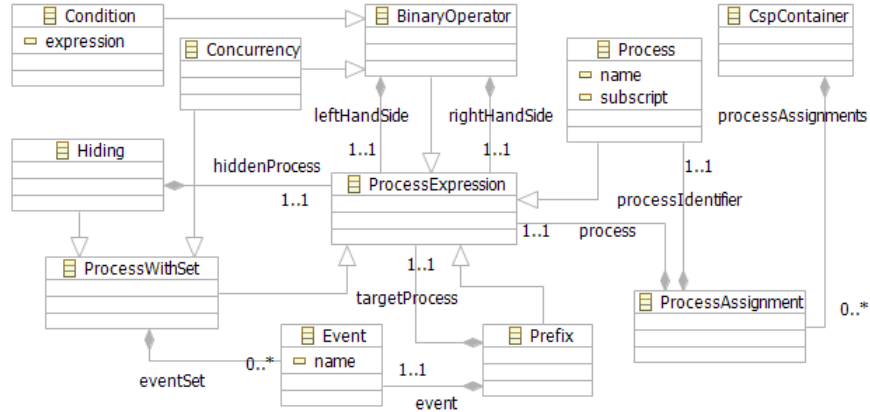


Fig. 2. CSP Metamodel

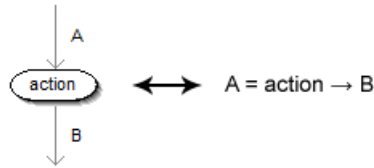
operator, a *BinaryOperator* combining two expressions or can be associated with a set of events (*ProcessWithSet*).

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 $E \parallel F$ (perform E and F in lock-step synchronisation of shared events). According to [2], the operator $E \llcorner b \nearrow F$ 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 the process *SKIP* represents successful termination, while the process *STOP* is a deadlock.

3 Transformation Method

We illustrate in this section the method of the desired transformation rules by showing intuitive correspondences between the two 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.

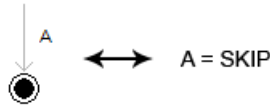
1. An *ActivityEdge* corresponds to a *ProcessIdentifier* while an *Action* to an *Event*. Without loss of generality we restrict Action nodes to have only one incoming and one outgoing edge.



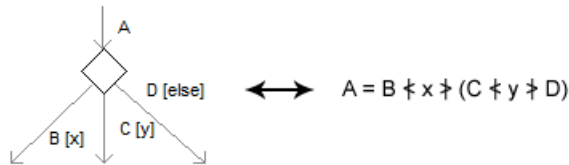
2. *InitialNode* corresponds to the first process assignment.



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

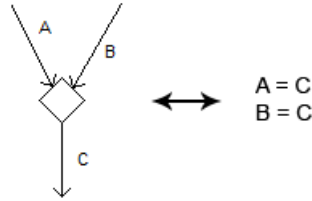


4. A *DecisionNode* corresponds to embedded *Condition* operators with the *guards* as their *condition expressions*.

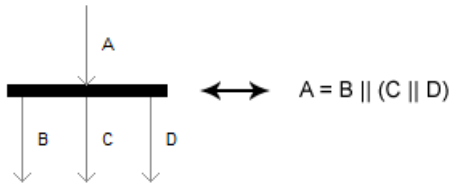


Note that this correspondence, which creates non-determinism at the syntactic level, leads to semantically equivalent processes. According to [1], *the order in which guards are evaluated is undefined and the modeller should arrange that each token only be chosen to traverse one outgoing edge, otherwise there will be race conditions among the outgoing edges*. Hence, if guard conditions are disjoint, syntactically different nestings are semantically equivalent.

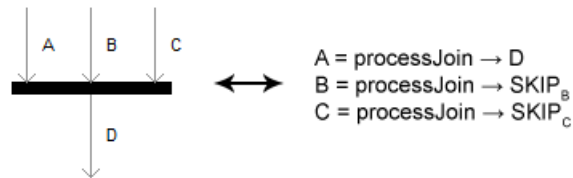
5. The *MergeNode* is mapped to an equation identifying the processes corresponding to the two incoming edges.



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 synchronisation 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 synchronisation 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*.

4 Challenges for the Approach

In this section we summarise the various requirements, the transformation tool should fulfill.

The transformation tool should 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 is important. Verification techniques are desirable, however not required. Termination and determinism up to process equivalence should be verified.

5 Example

In this section, we introduce an example activity diagram with a transformation result, that can be used to test and validate the transformation.

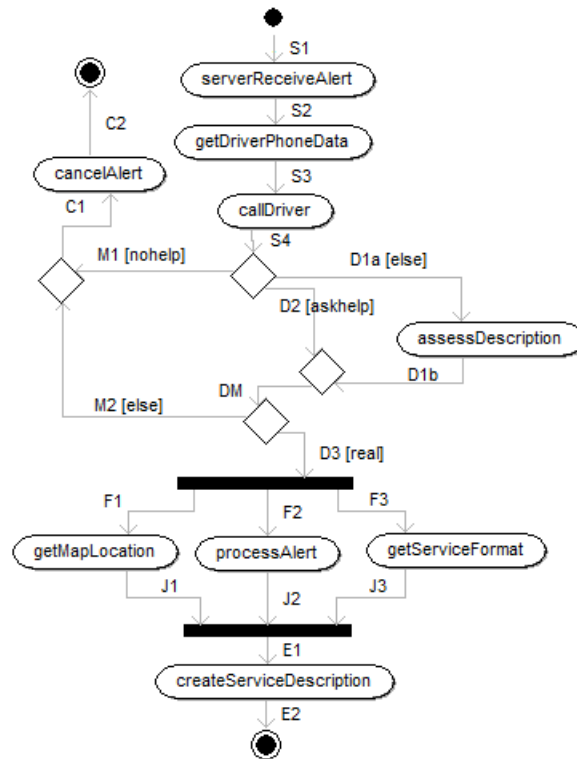


Fig. 3. Example Activity Diagram

The example activity diagram is shown in Figure 3. A possible result (up to process equivalence) of the transformation is the following.

$S1 = serverReceiveAlert \rightarrow S2$
 $S2 = getDriverPhoneData \rightarrow S3$
 $S3 = callDriver \rightarrow S4$
 $S4 = M1 \not\leftarrow nohelp \not\leftarrow (D2 \not\leftarrow askhelp \not\leftarrow D1a)$
 $M1 = C1$
 $D2 = DM$
 $D1a = assessDescription \rightarrow D1b$
 $D1b = DM$
 $DM = D3 \not\leftarrow real \not\leftarrow M2$
 $M2 = C1$
 $C1 = cancelAlert \rightarrow C2$
 $C2 = SKIP$
 $D3 = F1 \parallel F2 \parallel F3$
 $F1 = getMapLocation \rightarrow J1$
 $F2 = processAlert \rightarrow J2$
 $F3 = getServiceFormat \rightarrow J3$
 $J1 = processJoin \rightarrow E1$
 $J2 = processJoin \rightarrow SKIP$
 $J3 = processJoin \rightarrow SKIP$
 $E1 = createServiceDescription \rightarrow E2$
 $E2 = SKIP$

References

1. OMG: Unified Modeling Language, version 2.1.1. (2006)
<http://www.omg.org/technology/documents/formal/uml.htm>.
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice Hall (April 1985)
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: A System of Patterns. 1st edn. Volume Volume 1 of Pattern-Oriented Software Architecture. John Wiley and Sons (August 1996)