Electronic Communications of the EASST
Volume 6 (2007)

Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)

Rule-Level Verification of Business Process Transformations using CSP

Dénes Bisztray, Reiko Heckel

13 pages

Guest Editors: Karsten Ehrig, Holger Giese
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Rule-Level Verification of Business Process Transformations using CSP

Dénes Bisztray[1], **Reiko Heckel**[2]

Department of Computer Science, University of Leicester

[1] dab24@mcs.le.ac.uk, [2]reiko@mcs.le.ac.uk

Abstract: Business Process Reengineering is one of the most widely adopted techniques to improve the efficiency of organisations. Transforming process models, we intend to change their semantics in certain predefined ways, making them more flexible, more restrictive, etc.

To understand and control the semantic consequences of change we use CSP to capture the behaviour of processes before and after the transformation. Formalising process transformations by graph transformation rules, we are interested in verifying semantic properties of these transformations at the level of rules, so that every application of a rule has a known semantic effect.

It turns out that we can do so if the mapping of activity diagrams models into the semantic domain CSP is compositional, i.e., compatible with the embedding of processes into larger contexts.

Keywords: Business Process Reengineering, Activity Diagrams, Graph Transformation, CSP, Verification

1 Introduction

A *business process* is a flow of actions, representing the work of an individual, internal system, or external partner company, towards a definite business goal [Hav05]. A *business process model* is a specification of a set of business processes. Of the different aspects addressed by such models we will be interested in the behavioural one, the *workflow specification*.

When organisations adapt to new markets or optimise their business processes, their workflows need to change, too. Business Process Reengineering is concerned with the systematic analysis and redesign of business process models. Depending on the objectives, changes may apply to the internal structure of the process model, preserving its semantics, or to the behaviour itself, making the workflow more flexible or more restrictive, adding new features, etc.

Workflows are often modelled diagrammatically, e.g., by UML activity diagrams [OMG05]. Their semantics, the collection of all workflows conforming to the model, can be formalised as a (potentially infinite) set of action sequences, called *traces*. Rather than specifying these sets explicitly, one can define a mapping into a process calculus like CSP (Communicating Sequential Processes [Hoa85]). This allows the use of theories and tools for analysing properties of processes [Ros97]. In particular it becomes possible to check if a process transformation has the

desired semantic effect, e.g., if the processes before and after the change are equivalent, if one is a restriction or extension of the other.

However, if the workflow models are sufficiently complex, a complete formal analysis of the corresponding CSP processes may be impossible or too costly. Therefore, we are are looking for a formalisation of *local* process transformations, which can be analysed separately for their semantic effect and sequentially composed in order to implement more complex changes.

To define process transformations, we formalise the abstract syntax of activity diagrams in terms of typed graphs, where a type graph TG plays the role of a metamodel for the language and instance graphs G typed over TG represent individual diagrams. Changes to these diagrams can then be specified by typed graph transformation rules, providing us with a formalisation of local workflow redesign steps.

Hence our approach combines two main ingredients: CSP as a semantic domain and analysis technique for workflow models, and rule-based graph transformations for specifying local redesign steps. The main question is about the relation of one with the other: How does a redesign transformation affect the semantics of processes?

It turns out that the question can be answered at the level of rules if the mapping from activity diagrams to CSP is compositional in the sense that the mapping of a sub-activity diagram yields a sub-CSP process at the semantic level. Describing the mapping from activity diagrams to CSP by triple graph grammars [Sch94], we can use results from the theory of graph transformation to verify the compositionality of the mapping.

The paper is structured as follows: in Section 2, we informally introduce the concept of semantics based verification of business process redesign. In Section 3 we present the mapping of activity diagram to CSP. Section 4 formally states the requirements needed to verify the semantic effect of transformations at the rule level and discusses them with respect to the mapping defined before. Section 5 concludes the paper.

2 Business Process Reengineering

For a motivating example, we consider a simple transformation on a workflow that describes the unpacking of a notebook computer.

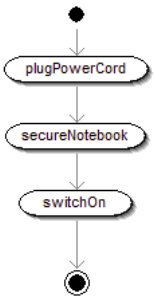


Figure 1: Sample process of unpacking a notebook

As Figure 1 illustrates, the workflow consists in three steps. First, we have to plug in the power

cord. Then, we secure the notebook with a cable lock. Finally, we have to switch on the computer. Analysing this process, we discover that this process could be made more flexible. For reacting to a situation where the cable lock is not available in time we may decide that the notebook can be switched on independently from securing it. The new process model, which allows both activities in either order as shown in Figure 2, represents a semantic extension of the old one.

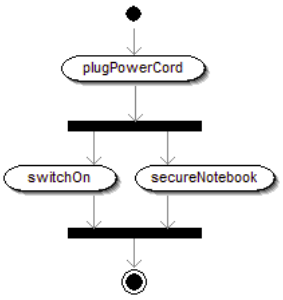


Figure 2: Redesigned notebook unpacking

Formally, the semantics of these processes is defined by standard CSP expressions. Up to substitution of process equations, the result of the mapping introduced in detail in Section 3 is shown below, with definition (1) describing the original process and equation (2-4) for the redesigned process.

$$UNPACK_O = plugPowerCord \rightarrow secureNotebook \rightarrow switchOn \rightarrow SKIP_{UNPACK_O}. \quad (1)$$

$$UNPACK_R = plugPowerCord \rightarrow (S_1 \parallel S_2) \quad (2)$$

$$S_1 = secureNotebook \rightarrow processJoin \rightarrow SKIP_{S_1} \quad (3)$$

$$S_2 = switchOn \rightarrow processJoin \rightarrow SKIP_{S_2} \quad (4)$$

$SKIP_A$ is defined as a process which does nothing but terminating successfully, with alphabet $A \cup \{\checkmark\}$ [Hoa85].

Analysing the processes before and after the transformation, we discover that

- the original process has only one trace $\langle plugPowerCord,\ secureNotebook,\ switchOn \rangle$;
- ignoring the *processJoin* action, in the redesigned process there are the two traces $\langle plugPowerCord,\ switchOn,\ secureNotebook \rangle$, $\langle plugPowerCord,\ secureNotebook,\ switchOn \rangle$.

The behaviour of the old process is thus present in the new one, i.e., $traces(UNPACK_O) \subseteq traces(UNPACK_R)$.

If the activity of unpacking a notebook is embedded into a larger process (e.g., setting up a workplace), it should be possible to derive the global consequences from the changes made to the smaller process. More generally, we want to predict the semantic effect of an operation before even performing it on the real (and potentially large) process. Thus, we formalise the change by

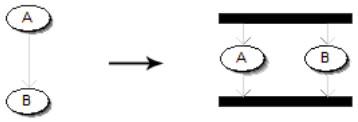


Figure 3: Redesign rule

graph transformation rules as the one sketched in Figure 3 and apply the mapping to CSP on its left- and right-hand side.

$$PROC_L = A \rightarrow B \rightarrow SKIP_{PROC_L} \tag{5}$$

$$PROC_R = (A \rightarrow processJoin \rightarrow SKIP_A \parallel B \rightarrow processJoin \rightarrow SKIP_B) \tag{6}$$

We can indeed observe that, after hiding the *processJoin* action, the traces of $PROC_L$ are included in those of $PROC_R$, and from general results about CSP trace refinement [Hoa85] it follows that this relation is closed under the embedding of CSP processes into context. To benefit from this fact we have to formalise and study the mapping of activity diagrams to CSP, which is described in the following section by means of triple graph grammars.

3 Mapping Activity Diagram to CSP

This section specifies a mapping from activity diagrams to CSP processes. Contrary to previous approaches [LBC00], we follow the approach of triple graph grammars (TGGs), where the abstract syntax of both activity diagrams and CSP processes are represented by typed graphs, i.e., instances of corresponding metamodels. A third metamodel as depicted in Figure 4, is used to capture the relation between corresponding elements of diagrams and processes. A brief introduction to TGGs is provided, however the detailed explanation can be found in [Sch94].

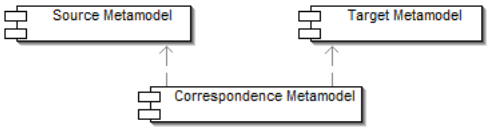
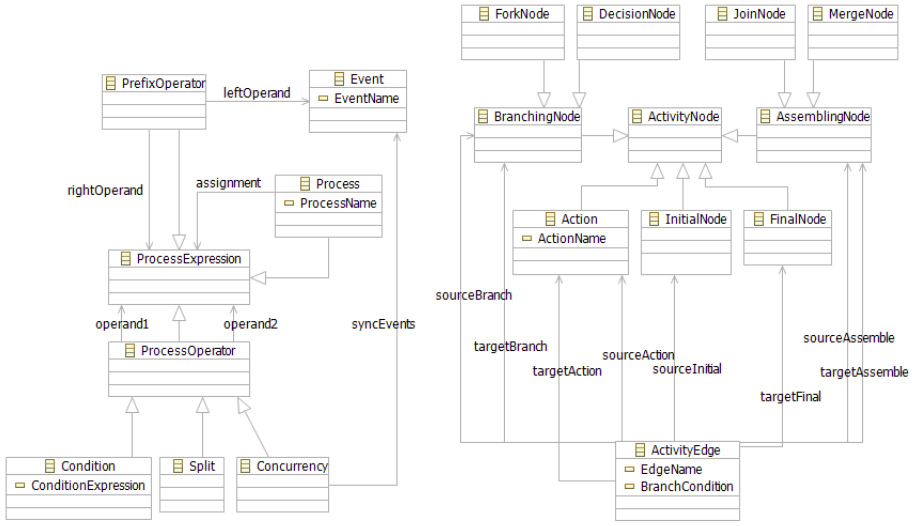


Figure 4: Triple graph grammar concept

3.1 Abstract Syntax

The metamodel for CSP processes, as far as required for our mapping rules, is shown in Figure 4(a). Following the Composite Pattern [BMR+96], a Process Expression either represents a Prefix $(x \rightarrow E)$ of a basic Event x followed by expression E, a Process equation $P = E$ assigning an expression E to a process name P, or a binary Process Operator combining two expressions.



(a) Abstract syntax for CSP (b) Abstract syntax for Activity Diagram

Figure 5: Abstract Syntax

- If b is a Boolean and E and F are process expressions, a Condition is an expression $E \not< b \not> F$ (*if b then E else F*);
- if E and F are expressions, Split is their sequential composition $E;F$ (*upon termination of E, continue as F*);
- 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*).

A simplified metamodel for activity diagram based on [OMG05] is shown in Figure 4(b). Nodes not present in the standard document include the BranchingNode and AssemblingNode. The BranchingNode is the superclass of the fork-like nodes that have one incoming edge and several outgoing edges. The AssemblingNode is the superclass for the join-like nodes, that have several incoming edges and one outgoing edge. Without loss of generality we restrict Action nodes to have only one incoming one outgoing edge.

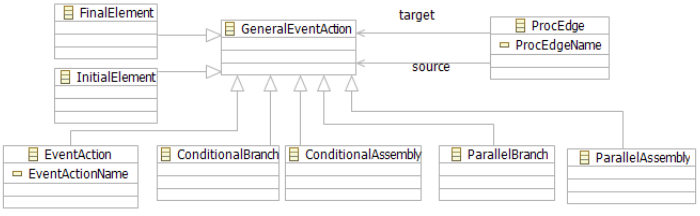


Figure 6: Correspondence metamodel

In order to be able to specify triple graph grammar rules, we require a correspondence metamodel as given in Figure 6. The EventAction is connected via associations to both Event in the Activity Diagram metamodel and Action in the CSP metamodel. The same is assumed for ProcEdge as the intermediary of Process and ActivityEdge, etc. Such associations are omitted from the illustrations to simplify the layout.

3.2 Transformation Method

Next we illustrate the design of our transformation rules, concentrating on a single rule for a detailed representation while using a semi-formal notation based on the concrete syntax for the others.

Consider the following simple example rule for transforming an Action node. The concrete syntax of the transformation rule is depicted in Figure 7.

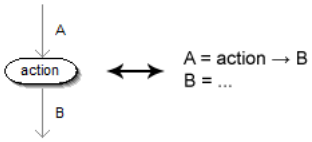


Figure 7: Action rule with concrete syntax

The idea behind the mapping is to relate an Edge in the activity diagram to a Process name in CSP. A previously introduced edge/process name *A* is defined in terms of a new prefix expression, and the continuation edge/process name *B* is introduced.

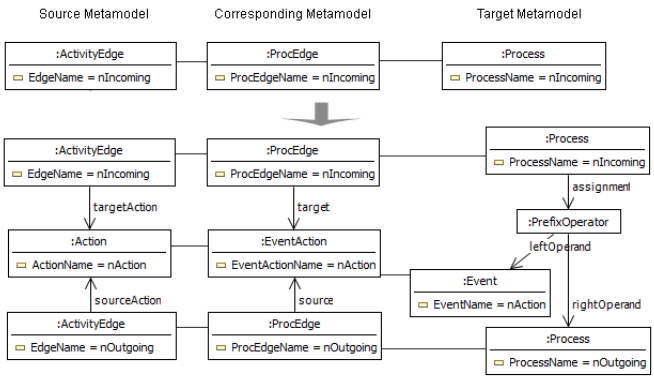


Figure 8: TGG Rule

The formal TGG rule is shown in Figure 8, to be read from top to bottom. In the top (the left-hand side of the rule) a triplet of ActivityEdge, ProcEdge and Process is matched. The bottom (right-hand side) generates simultaneously the Action and outgoing ActivityEdge of the activity diagram, the PrefixOperator with the Event and continuation Process and the relational elements between them.

In order to be used for a transformation from activity diagrams to CSP (rather than a symmetrical specification of their correspondence), the translation of the TGG rule to ordinary graph transformation rule is illustrated in Figure 9. There are various tools that enable the automatic generation of graph transformation rules from TGG rules. We match the pattern of an Action with incoming and outgoing edges, and create the corresponding CSP elements. A negative application condition [EEPT06] is defined for the relational element to prevent us from applying the same rule twice.

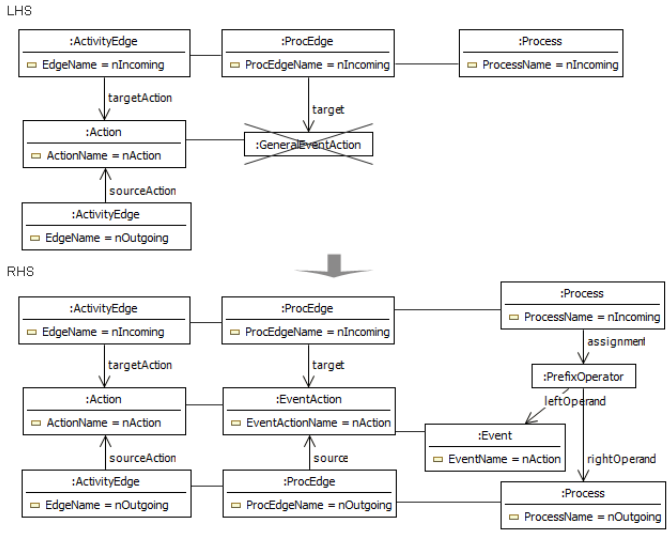


Figure 9: Graph Transformation rule

Indeed, a pragmatic benefit of TGGs consists in the use of the correspondence model for controlling the progress of transformation. The creation of relational elements, in combination with negative application conditions on the correspondence and target model, allow us to retain the original model and restrict ourselves to non-deleting rules. These two properties will be important later.

3.3 Transformation Rules

In this section we define the remaining transformation rules based on the concrete syntax of CSP and activity diagrams.

First, we consider the InitialNode depicted in Figure 10. Although this node is not mapped to anything directly, its outgoing ActivityEdge is related to a process definition with the same name. This will be the first process.

The transformation of a DecisionNode depicted in Figure 12 is a more complicated case. The concrete syntax is obvious, but Condition is a binary operator. Thus, we have to build a binary tree bottom-up as depicted in Figure 11. First we match the *else* branch with an arbitrary edge and create the lowest element of the tree. Then we build a tree adding the elements one-by-one, gluing its top to the Process that represents the incoming edge.

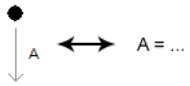


Figure 10: InitialNode

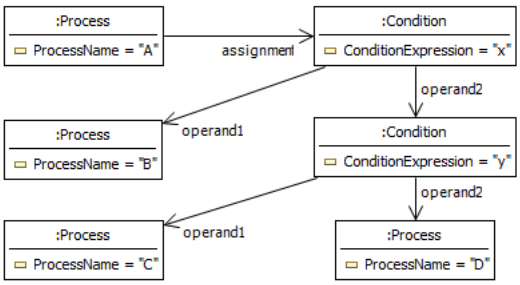


Figure 11: Abstract syntax tree for the result of DecisionNode transformation

Note that this transformation, which creates non-determinism at the syntactic level, leads to semantically equivalent processes. According to [OMG05], *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.

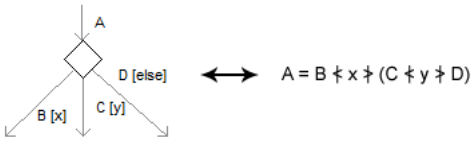


Figure 12: DecisionNode

The MergeNode is a simpler case, as illustrated in Figure 13. It is mapped to an equation identifying the processes corresponding to the two incoming edges.

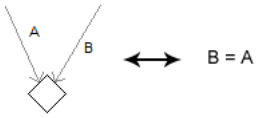


Figure 13: MergeNode

ForkNode and JoinNode represent the most complex cases. Before describing the transformation, 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.

The rule for the ForkNode is shown in Figure 14. The Concurrency operator is binary, so by processing the nodes one-by-one, we create a binary abstract syntax tree of Concurrency nodes like we did in Figure 11 for the Condition operator. Since $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$, the different trees are semantically equivalent.

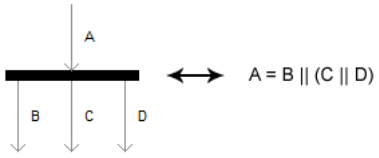


Figure 14: ForkNode

The transformation of JoinNode is depicted in Figure 15. The first edge that meets the JoinNode is chosen to carry the continuation process, while the others terminate in a *SKIP*. As we mentioned, CSP expressions 2-4) are based on this transformation. The process corresponding to D is substituted, since it terminates after the syncronisation.

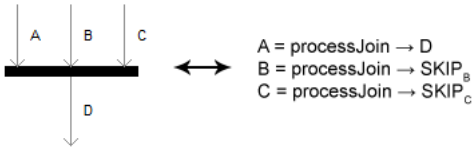


Figure 15: JoinNode

4 Rule-Level Verification

In this section we provide a formalisation of the notions required to define the compositionality condition of the mapping and state the main objective of the approach as a theorem. As mentioned in Section 1, we handle business processes as typed graphs, and reengineering steps as graph transformations. Since the contributions in this section are based on equivalence and refinement of CSP processes, we summarise the necessary definitions based on [Hoa85].

A *trace* is a finite sequence of symbols recording the events in which the process has engaged up to some point in time. The set of all traces of a process P is denoted by $traces(P)$. Processes P and Q are trace equivalent ($P \equiv Q$) if $traces(P) = traces(Q)$. P is a refinement of Q ($P \sqsubseteq Q$) if $traces(P) \subseteq traces(Q)$. A context is a process expression $E(X)$ with a single occurrence of a

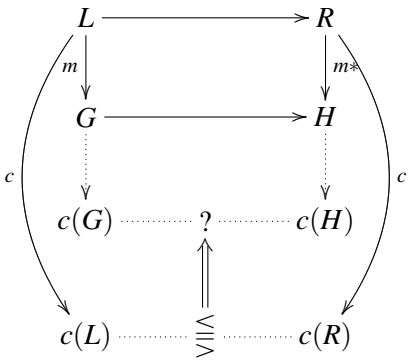


Figure 16: CSP correspondence for behaviour verification

distinguished process variable X. The relations of trace equivalence and refinement are closed under context, i.e., $P \equiv Q \implies E(P) \equiv E(Q)$ and $P \sqsubseteq Q \implies E(P) \sqsubseteq E(Q)$.

Denoting the mapping from activity diagrams to CSP by c, the idea of rule-level verification is illustrated in Figure 16. The original business process is given by graph G, the redesigned one by the resulting graph H of the application of rule $p : L \to R$ at match m and comatch m^*. Applying the mapping c to the rule's left- and right hand side, we compare the corresponding CSP expressions $c(L)$ and $c(R)$, checking, e.g., that $c(L) \sqsubseteq c(R)$ (the right process has more traces than the left one). From this relation at the level of the rule we hope to conclude that the same holds for all its transformations, i.e., $c(G) \sqsubseteq c(H)$. For such a property to hold, we make the following assumption on the mapping c.

Definition 1 (compositionality) A mapping c from graphs to CSP expressions is compositional if for all injective graph morphisms $m : L \to G$ there exists a context E such that $c(G) \equiv E(c(L))$. Moreover, this context is uniquely determined by the part of G not in the image of L, i.e., given a pushout diagram as below with injective morphisms only, and a context F with $c(D) \equiv F(c(K))$, then E and F are equivalent.

$$\begin{array}{ccc} K & \xrightarrow{l} & L \\ {\scriptstyle d}\downarrow & & \downarrow{\scriptstyle m} \\ D & \xrightarrow{g} & G \end{array}$$

Definition 1 applies particularly where L is the left hand side of a rule and G is the given graph of a transformation. In this case, the CSP expression generated from L contains the one derived from G up to equivalence, while the context is uniquely determined by $G \setminus m(L)$.

Theorem 1 *If a mapping c from graphs to CSP expressions is compositional, for all transformations $G \overset{p,m}{\Longrightarrow} H$ via rule $p : L \to R$ with injective match m, it holds that $c(L) \triangleq c(R)$ implies $c(G) \triangleq c(H)$, where $\triangleq$ is any relation in $\{\equiv, \sqsubseteq, \sqsupseteq\}$.*

Proof. By assumption the match m, and therefore the comatch $m^* : K \to H$ are injective. Since the mapping c is compositional, according to Definition 1 there are contexts E and F such that

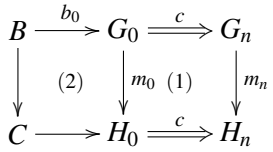


Figure 17: Extension diagram

$c(G) \equiv E(c(L))$ and $c(H) \equiv F(c(R))$. Now, $E(c(L)) \triangleq E(c(R))$ since $c(L) \triangleq c(R)$ and the relation is closed under context. Finally, $E(c(R)) \equiv F(c(R))$ by the uniqueness of the context. □

In the following we discuss how our mapping from activity diagram to CSP satisfies the compositionality condition. Since we did not introduce our mapping formally, we only provide a sketch of a proof. The main argument is based on the Embedding Theorem [Ehr77] and its extension to conditional graph transformations [Hec95]. In the basic version we assume a graph H_0 including a smaller one G_0 with inclusion morphism m_0. For a sequence of transformation $c: G_0 \stackrel{*}{\Longrightarrow} G_n$ we create a boundary graph B and a context graph C. The boundary graph is the smallest subgraph of G_0 which contains the identification points and dangling points of m_0. Since (2) is a pushout, the context graph can be determined. If none of the productions of c deletes any item of B, then m_0 is consistent with c and there is an extension diagram over c and m_0. This basically means that H_n is the pushout complement of c and m_0, thus can be determined without applying the transformation c on H_0. Hence, the compositionality condition holds for c.

Adding context and boundary graph to our picture, we get the extension diagram in Figure 17. The initial graph G_0 is either the LHS or the RHS of a production rule in the reenginering transformation. With our previous notation, G_n equals $c(G_0)$, i.e., our graph transformation implements the mapping c and our inclusion graph morphism is the match m_0 for the actual rule.

In our case, consistency of the embedding is trivial because, due to the use of triple graph grammars, our rules never delete nodes. Moreover, the embeddings we are interested in only add new context to the source model part of the graph, while negative conditions are only concerned with the correspondence and target part. This ensures that the embedding never violates any of the application conditions, i.e., the embedding theorem holds for the conditional transformations, too [Hec95].

Uniqueness of the context follows from the fact that the mapping c is deterministic.

5 Conclusion

In this paper we have studied the relation between two dimensions of model transformations: a semantic dimension representing a mapping of models into another formalism, and a dimension of change capturing the evolution of models. We have investigated an example and defined a condition which ensures that the effect of change on the semantics of models is predictable at the level of rules.

The approach differs from previous work on semantics-preserving transformation [BEH06] by the use of denotational rather than operational semantics. This makes is necessary to introduce a “semantic” formalism like CSP, but it allows the use of theories and tools of the semantic

domain for analysing models. A similar approach has been proposed in [EGHK02], but without formalising it.

Future work consists in completing the definition of the mapping and the proof that it satisfies the compositionality condition. In relation to business processes we intend to add a concept of observable vs. hidden actions to make more flexible use of existing notions of process equivalence.

Acknowledgements: This work has been partially sponsored by the project SENSORIA, IST-2005-016004. The authors also wish to thank Hartmut Ehrig for pointing out an improvement of the main theorem of the paper.

Bibliography

[BEH06] L. Baresi, K. Ehrig, R. Heckel. Verification of model transformations: A case study with BPEL. In Bruni et al. (eds.), *Proc. 2nd Symposium on Trustworthy Global Computing (TGC 2006), November 2006, Lucca, Italy*. LNCS. November 2006.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns*. Pattern-Oriented Software Architecture Volume 1. John Wiley and Sons, 1st edition, August 1996.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EGHK02] G. Engels, L. Groenewegen, R. Heckel, J. Küster. Consistency-preserving model evolution through transformations. In Jezequel et al. (eds.), *Proc. UML 2002, Dresden, Germany*. LNCS 2460. Springer-Verlag, Oct. 2002.

[Ehr77] H. Ehrig. Embedding Theorems in the Algebraic Theory of Graph Grammars. In *Fundamentals of Computation Theory, Proceedings of the 1977 International FCT-Conference*. Pp. 245–255. Poznan-Kórnik, Poland, September 1977.

[Hav05] M. Havey. *Essential Business Process Modeling*. Theory In Practice. O'Reilly Media, August 2005.

[Hec95] R. Heckel. Embedding of Conditional Graph Transformations. In Valiente Feruglio and Rosello Llompart (eds.), *Proc. Colloquium on Graph Transformation and its Application in Computer Science*. 1995.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, April 1985.

[LBC00] G. Luttgen, M. von der Beeck, R. Cleaveland. A compositional approach to statecharts semantics. In *Foundations of Software Engineering*. Pp. 120–129. 2000.

[OMG05] OMG. Unified Modeling Language, version 2.0. Website, August 2005. http://www.omg.org/technology/documents/formal/uml.htm

[Ros97] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1st edition, November 1997.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Tinhofer (ed.), *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*. LNCS 903, pp. 151–163. Springer-Verlag, 1994.