Electronic Communications of the EASST
Volume 16 (2009)

Proceedings of the Doctoral Symposium at the International Conference on Graph Transformation (ICGT 2009)

Verification of Architectural Refactorings: Rule Extraction and Tool Support

Dénes Bisztray, Reiko Heckel and Hartmut Ehrig

16 pages

Guest Editors: Andrea Corradini, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verification of Architectural Refactorings: Rule Extraction and Tool Support

Dénes Bisztray[1], Reiko Heckel[2] and Hartmut Ehrig[3]

[1] dab24@mcs.le.ac.uk
[2] reiko@mcs.le.ac.uk
Department of Computer Science
University of Leicester
United Kingdom

[3] ehrig@cs.tu-berlin.de
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
Germany

Abstract: Software in use needs to be adapted to changing requirements, otherwise it becomes obsolete. Often, this involves changing the architecture of the system. To avoid the introduction of unwanted or removal of desired behaviour, these changes need verification. While verifying large systems consumes considerable resources, the verification of only the changed parts can, under certain conditions, give the required assurance. This opens the possibility of creating formally verified, reusable refactoring patterns. However, a mechanism for extracting such patterns is needed. To address this problem, a theoretical framework is presented that allow to formally reason about the rule extraction process. In order to harness the theoretical results, a visual editor and tool chain are introduced to aid developers in extracting refactoring rules and prove their behavioural correctness.

Keywords: Graph Transformations, Refactoring, Semantics

1 Introduction

With the recent success of the component-based and service-oriented paradigm, the complexity of software also increased. To tackle complexity, architectural models aid the developers. However, a software in constant use, must continually evolve, otherwise it becomes progressively less satisfactory [Leh96]. During the adaptation to changed requirements and improvement of internal structure, changes may be required that preserve the observable behaviour of the systems. In OO programming, such behaviour-preserving transformations are called refactorings [FBB+99].

For distributed and service-oriented applications, the important changes take place at the level of architectural models. These changes have to be checked for behaviour preservation. To avoid the costly verification of refactoring steps on large systems, we extract a (usually much smaller) rule from the transformation performed and verify this rule instead. However, the notion of observable behaviour has to be established, formal requirements for extracting the refactoring rule and methods that can verify its behaviour preservation are required.

The method of verifying architectural refactorings consists of three ingredients: the modelling language used, its semantics, and the relation capturing our idea of behaviour preservation. For representing the type and instance-level architecture of our system, we use UML *component* and *composite structure diagrams* in conjunction with *activity diagrams* specifying the workflows executed by component instances as described in [BHE08].

To verify the semantic relation between source and target models, the behaviour of the combined structure diagram is formalised by denotational semantics, using Communicating Sequential Processes (CSP) [Hoa85] as semantic domain. CSP is a process algebra for concurrent systems.

A mapping *sem* has been defined from the UML diagrams to CSP processes by means of graph transformation rules. The *semantic relation* of behaviour preservation can conveniently be expressed using one of the refinement and equivalence relations on CSP processes.

As mentioned, it would be advantageous if we could focus our verification on those parts of the model that have been changed, that is, verify the refactoring *rules* rather than the actual steps. This is indeed possible, as we have shown in [BHE08]: assuming a refactoring $G \Longrightarrow H$, via the graph transformation rule $p : L \to R$, if the relation $\mathscr{R}$ holds for $sem(L)\ \mathscr{R}\ sem(R)$ then $sem(G)\ \mathscr{R}\ sem(H)$ also holds. This is feasible due to the compositionality property [BHE08] of the semantic mapping.

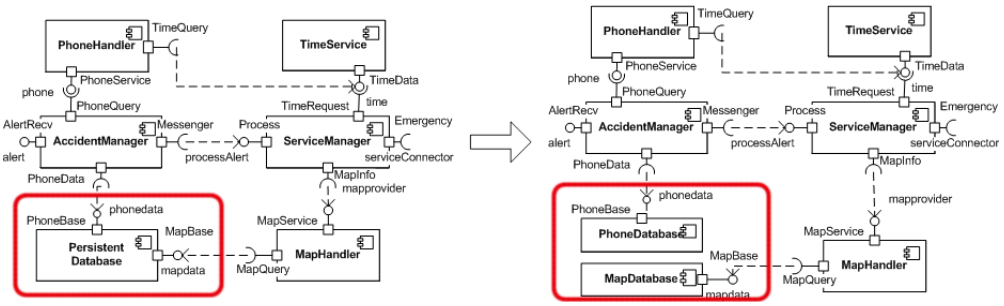


Figure 1: Example Refactoring

An example refactoring is depicted in Figure 1. As the interfaces are unchanged and the behaviour is encapsulated inside the components, the marked parts can be extracted. Then, this transformation rule can be verified instead of the complete system. Although this extraction was obvious, there can be complicated refactorings that span change on the behaviour of multiple components. To determine the mechanics of producing a rule, we perform extractions on proven and successful refactorings. In general for a transformation $G \Longrightarrow H$ with $sem(G)\ \mathscr{R}\ sem(H)$, we want to extract the smallest rule such that:

1. when applying it on G at the appropriate match, the transformation step produces H
2. $sem(L)\ \mathscr{R}\ sem(R)$

The rule extraction consists of two steps. In the first step, the difference between G and H is extracted, to form a minimal production rule. The presented method uses initial pushouts [EEPT06],

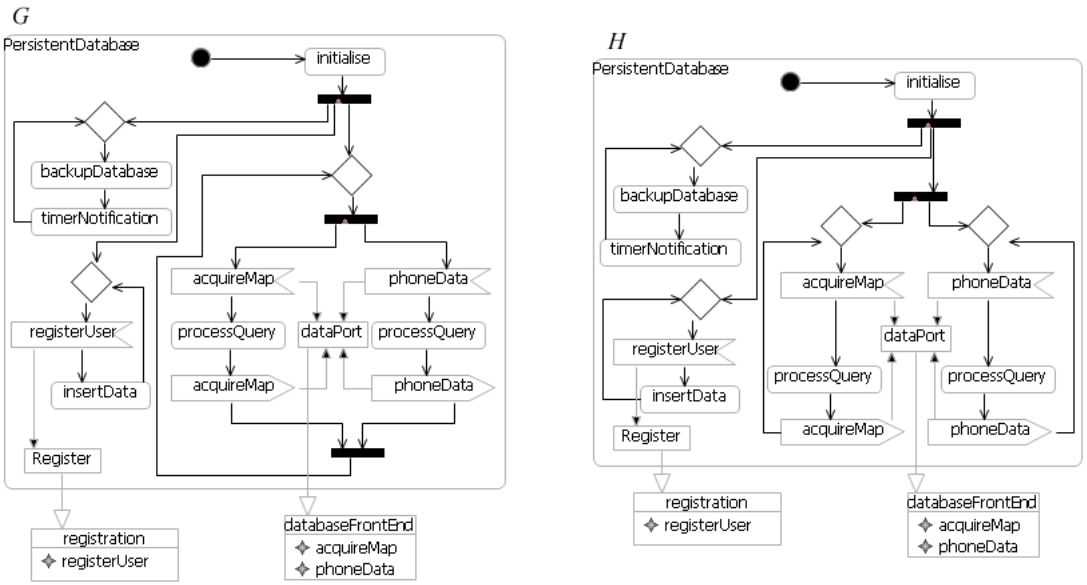


Figure 2: The original and refactored system

and is shown to fulfill *requirement 1*. As the minimal rule may represent a semantically incomplete part of the model, often, a context needs to be added to fulfill *requirement 2*. This context is determined interactively by the refactoring developer. Being an iterative process, tool support is needed not only for verifying complicated system refactorings but also for supporting the rule extraction mechanisms.

The paper is organised as follows. In Section 2 we present our methodology with the help of a case study. Then, we establish a formal framework for the rule extraction process. The extraction of the minimal rule is presented in Section 3, the method to include the necessary context is in Section 4. The reasoning behind the rule-level verification is presented in Section 5. The tool support for rule design is introduced in Section 6. And then, we conclude.

2 Methodology

To provide a general overview, we present in this section the rule extraction process using an example based on the *Car Accident Scenario* from the SENSORIA Automotive Case Study [WCG+06]. The extraction process consists of the following major steps:

1. The minimal graph rule is extracted that produces the refactored system when applied on the original one.

2. Necessary context is added to the minimal rule to make it semantically sound.

Our case study is based on the component *PersistentDatabase*, which is a simplified database component for a car accident server. The accident server is responsible for receiving distress

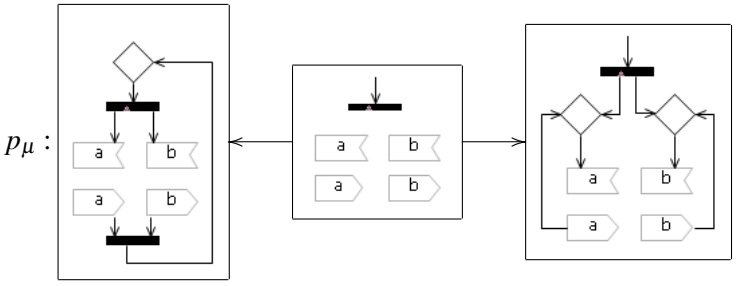


Figure 3: Extracted minimal rule

signals from cars. On an incoming signal it calls the driver to verify if the accident is real and ask the dispatch of emergency services if required. The database component contains the phone numbers of the registered users, and also the map of the area within jurisdiction. The accident server can query the phone number of registered users based on their identification or ask for a route plan from the nearest emergency service station to the location of the crashed car.

As can be seen in Figure 4, the activity diagrams describing behaviour, are embedded inside the components. We call this a *combined structure diagram*. Aside from the usual activity elements, there is a port in the component called *dataPort* which is connected to the *send signal* and *accept event actions*. This defines the communication channels precisely. The actions that an instance level channel may engage in are defined by the relevant *DataAccess* interface. As the port implements the interface, the first *mapQuery* accept event action receives the function call, and the corresponding send signal action sends the reply back.

The *PersistentDatabase* is a naive design: it synchronises after every phone number access with route query. This assumes that every distress signal is real, and thus needs emergency route plan. Fortunately, in several cases the crash may not need medical attention or may even be a false alarm. As the system needs more independency, the synchronisation node is deleted, and the two database engines work completely independently.

Extraction of the minimal graph transformation rule helps us to identify the changes in the system. The extracted minimal rule is shown in Figure 3. Minimality means that it is the smallest rule that produces the refactored system when applied on the original one at the appropriate match.

In most cases, including the present one, the minimal rule is not semantically sound. Moreover, it seems to suffer from syntactic errors (dangling edges). Note, however, that these diagrams are instances of metamodels where the edges are represented as nodes. To achieve the semantical completeness of our minimal rule, we have to address two problems.

1. It is possible to have a valid instance of the metamodel that is not semantically complete. A *send signal action* can mean both function call (if connected to a required interface) or replying the return value (if connected to a provided interface). In the minimal rule the *send signal action* and *accept event action* is not connected to any port or interface, leaving their meaning ambiguous.

2. The other problem is that the extracted minimal rule is not precisely what the refactoring intended to express. We assume that the fork and the join node form some kind of pair,

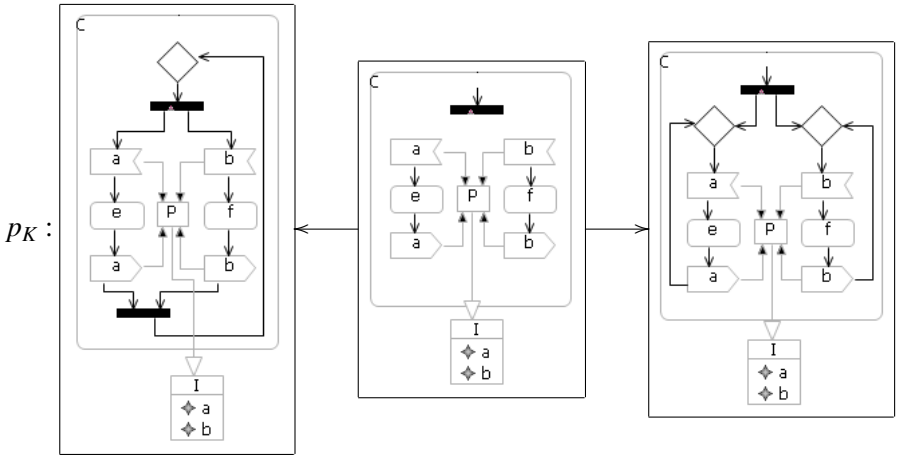


Figure 4: Extracted rule with added context

and as such they are connected to the same line of control flow. In the minimal rule, this is not expressed: the join node can synchronise with arbitrary flows in the system.

To overcome these faults, context needs to be included from the original system. It is important to ensure that the included context is the same on both sides.

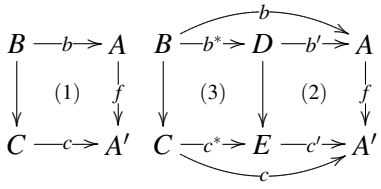
After the developer selected the necessary context, the resulting rule is shown in Figure 4. Verification shows it to be behaviour preserving. Thus, when the rule is used for refactoring, there is no further need for verifying the system. Since we proved the behaviour preservation of the rule, we may reuse it as a general refactoring pattern.

3 Extraction of Minimal Rule

The theoretical contributions are presented in two sections following the steps outlined in Section 2. In the present section, we introduce the construction that produces the minimal rule. Section 4 deals with step 2: it shows the context inclusion process. The process of minimal rule extraction assumes that only the original system G with refactored system H and their relation are given.

First, we introduce the concept of initial pushout, as it is used extensively throughout the paper. Initial pushout is a complement construction. The context graph C as shown in [EEPT06] is the smallest subgraph of A' that contains $A' \setminus f(A)$.

Definition 1 **(initial pushout** [EEPT06]) Given a morphism $f : A \to A'$, an injective morphism $b : B \to A$ is called the *boundary* over f if there is a pushout complement of f and b such that (1) is a pushout initial over f. Initiality of (1) over f means, that for every pushout (2) with injective b' there exists unique morphism $b^* : B \to D$ and $c^* : C \to E$ with injective b^* and c^* such that $b' \circ b^* = b$, $c' \circ c^* = c$ and (3) is a pushout. B is then called the boundary object and C the context with respect to f.



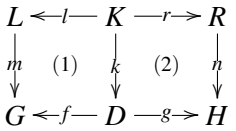
For clarity we present the definition of a *graph production* and a *graph transformation*.

Definition 2 **(graph production** [CMR+97]) A *(typed) graph production* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ also called *graph transformation rule* consists of (typed) graph L,K,R, called the left-hand side, gluing graph and the right-hand side respectively, and the two injective (typed) graph morphisms l and r.

Given a (typed) graph production p, the *inverse production* is defined by $p^{-1} = R \xleftarrow{r} K \xrightarrow{l} L$.

The difference between a *graph production* and a *typed graph production* (and thus between a *graph transformation* and *typed graph transformation*) is that the latter operates on typed graphs [Ehr87] while the former deals with simple graphs without any typing.

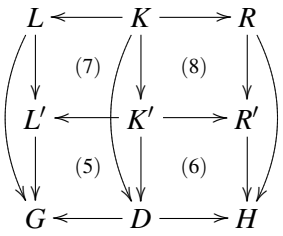
Definition 3 **(graph transformation** [CMR+97]) Given a (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a (typed) graph G with a (typed) graph morphism $m : L \to G$, called the match, a *direct (typed) graph transformation* $G \overset{p,m}{\Rightarrow} H$ from G to a (typed) graph H is given by the following double-pushout (DPO) diagram, where (1) and (2) are pushouts in the category *Graphs* (or $Graphs_{TG}$ respectively):



A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of direct (typed) graph transformations is called a *(typed) graph transformation* and is denoted by $G_0 \overset{*}{\Rightarrow} G_n$. For $n = 0$, we have the identical (typed) graph transformation $G_0 \overset{id}{\Rightarrow} G'_0$. Moreover, for n_0 we allow also graph isomorphisms $G_0 \cong G'_0$, because pushouts and hence also direct graph transformations are only unique up to isomorphism.

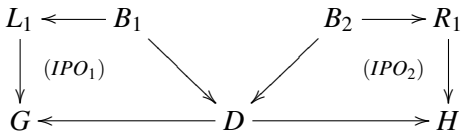
Minimality, as mentioned intuitively means the smallest rule, that produces H when applied on G. The formal definition is the following:

Definition 4 **(minimality)** A graph transformation rule $p : L \leftarrow K \rightarrow R$ is *minimal* over direct graph transformation $G \leftarrow D \rightarrow H$ if $K \rightarrow D$ is injective and for each rule $p' : L' \leftarrow K' \rightarrow R'$ with injective morphism $K' \rightarrow D$ and pushouts (5) and (6), there are unique $L \rightarrow L'$, $K \rightarrow K'$ and $R \rightarrow R'$ morphisms such that the following diagram commutes and (7), (8), $(5)+(7)$ and $(6)+(8)$ are pushouts.



Now, we present the construction that leads to the minimal rule.

Definition 5 (minimal rule) Given a span of injective graph morphisms $G \leftarrow D \rightarrow H$ with initial pushouts IPO_1 over $D \rightarrow G$ and IPO_2 over $D \rightarrow H$. The following construction will define transformation rule $p_\mu : L \leftarrow K \rightarrow R$ over $G \leftarrow D \rightarrow H$.

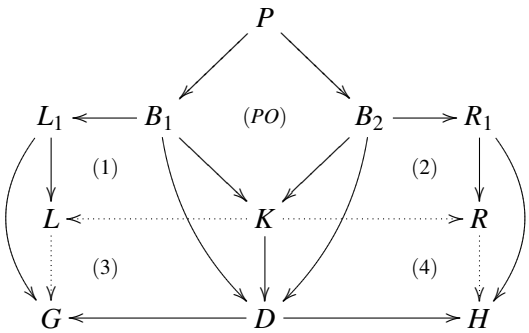


1. Define $B_1 \leftarrow P \rightarrow B_2$ as a pullback of $B_1 \rightarrow D \leftarrow B_2$ and $B_1 \leftarrow K \rightarrow B_2$ as a pushout of $B_1 \leftarrow P \rightarrow B_2$ with induced morphism $K \rightarrow D$.

2. Construct $L_1 \rightarrow L \leftarrow K$ as a pushout of $L_1 \leftarrow B_1 \rightarrow K$ with induced morphism $L \rightarrow G$. Similarly, $R_1 \rightarrow R \leftarrow K$ is a pushout of $R_1 \leftarrow B_2 \rightarrow K$ with induced morphism $R \rightarrow H$.

3. Since $IPO_1 = (1) + (3)$ and (1) is a pushout, because of the pushout decomposition property (3) is also a pushout. Similarly $IPO_2 = (2) + (4)$ and (2) being a pushout implies pushout (4).

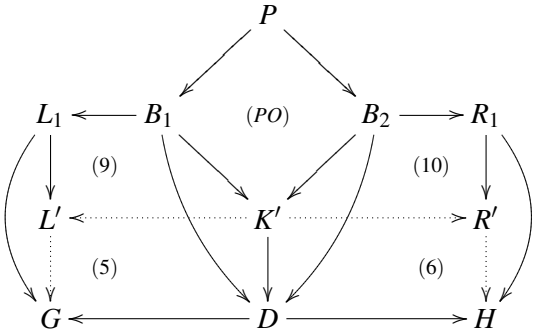
4. By the constructions of the initial pushouts, $B_1 \rightarrow D$ and $B_2 \rightarrow D$ are injective and hence also $P \rightarrow B_1$, $P \rightarrow B_2$, $B_1 \rightarrow K$, $B_2 \rightarrow K$ as well. This implies by the pushout and pullback properties that also $K \rightarrow D$ is injective.



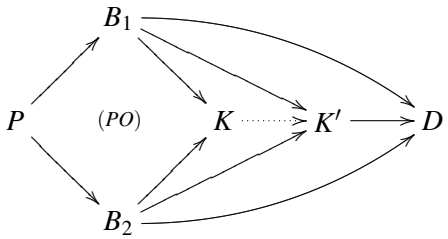
Then $L \leftarrow K \rightarrow R$ with pushouts (3) and (4) is p_μ over $G \leftarrow D \rightarrow H$ with injective morphisms $L \rightarrow G$, $K \rightarrow D$ and $R \rightarrow H$. Moreover injective $G \leftarrow D \rightarrow H$ implies injective $L \leftarrow K \rightarrow R$.

Theorem 1 ***(minimal rule)*** *Assuming a span of injective graph morphisms $G \leftarrow D \rightarrow H$, the graph transformation rule $p_\mu : L \leftarrow K \rightarrow R$ created according to Definition 5 is minimal.*

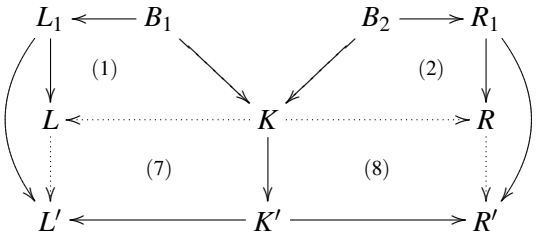
Proof. Given pushouts (5) and (6) over $G \leftarrow D \rightarrow H$ with injective morhpism $K' \rightarrow D$, we have by IPO_1 and IPO_2, unique $L_1 \rightarrow L'$, $B_1 \rightarrow K'$, $B_2 \rightarrow K'$ and $R_1 \rightarrow R'$ such that (9) and (10) are pushouts.



As $P \rightarrow B_1 \rightarrow D$ and $P \rightarrow B_2 \rightarrow D$ commutes and $K' \rightarrow D$ is an injective morphism, it implies that $P \rightarrow B_1 \rightarrow K'$ and $P \rightarrow B_2 \rightarrow K'$ also commutes and hence a unique $K \rightarrow K'$ morphism exists and the diagram below commutes.



Now pushouts (1) and (2) implies unique morphisms $L_1 \rightarrow L$ and $R_1 \rightarrow R$ such that the following diagram commutes and (7) and (8) are pushouts because of the pushout-decomposition of pushouts (9) and (10).



And also $L \rightarrow L' \rightarrow G$ commutes with $L \rightarrow G$ using the pushout properties of L and similarly $R \rightarrow R' \rightarrow H$ commutes with $R \rightarrow H$ using the pushout properties of R.

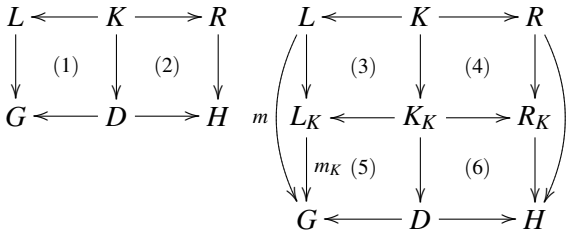
Uniqueness of $L \rightarrow L'$, $K \rightarrow K'$ and $R \rightarrow R'$ in the minimality diagram follows from the injectivity of $K' \rightarrow D$, $L' \rightarrow G$ and $R' \rightarrow H$. □

4 Inclusion of Necessary Context

As shown before, the sheer structural difference between the old and the new system is semantically not always comprehensible. To tackle this problem, the refactoring designer includes necessary context that makes the rule semantically complete. In Section 2 we shown two barriers to semantical completeness, however the list is not complete. The constraints that define the context is different each case and based on human intuition. Thus formalising a general semantical completeness is unlikely.

The following process describes the theoretical framework of the context inclusion.

Definition 6 (context inclusion process) Given minimal rule $p_\mu : L \leftarrow K \rightarrow R$ over $G \leftarrow D \rightarrow H$ with pushouts (1) and (2).



The context inclusion is defined by a suitable factorisation $K \rightarrow K_K \rightarrow D$ of $K \rightarrow D$ with injective $K_K \rightarrow D$, where L_K and R_K are defined by pushouts (3) and (4). By pushout decomposition this leads to pushouts (5) and (6) and $p_K : L_K \leftarrow K_K \rightarrow R_K$, where pushout $(1) = (3) + (5)$ and pushout $(2) = (4) + (6)$.

Hence we have $G \Rightarrow H$ via (p_K, m_K) by pushouts (5) and (6) with span $G \leftarrow D \rightarrow H$. Moreover the injectivity of $p_\mu : (L \leftarrow K \rightarrow R)$ implies that $p_K : (L_K \leftarrow K_K \rightarrow R_K)$ is injective as well.

The construction of the minimal and extracted rule based on the refactorisation $K \rightarrow K_K \rightarrow D$ of $K \rightarrow D$ using our database example is shown in Figure 5.

5 Towards Refactoring Design Patterns

As the aim of rule extraction is to verify only the refactoring rules, we present the theoretical background that enables such feat. In Section 2 we assumed that the refactoring on system level is behaviour preserving. As mentioned in Section 1, when using a compositional mapping to a semantic domain, the verification results of rules can substitute the system verification.

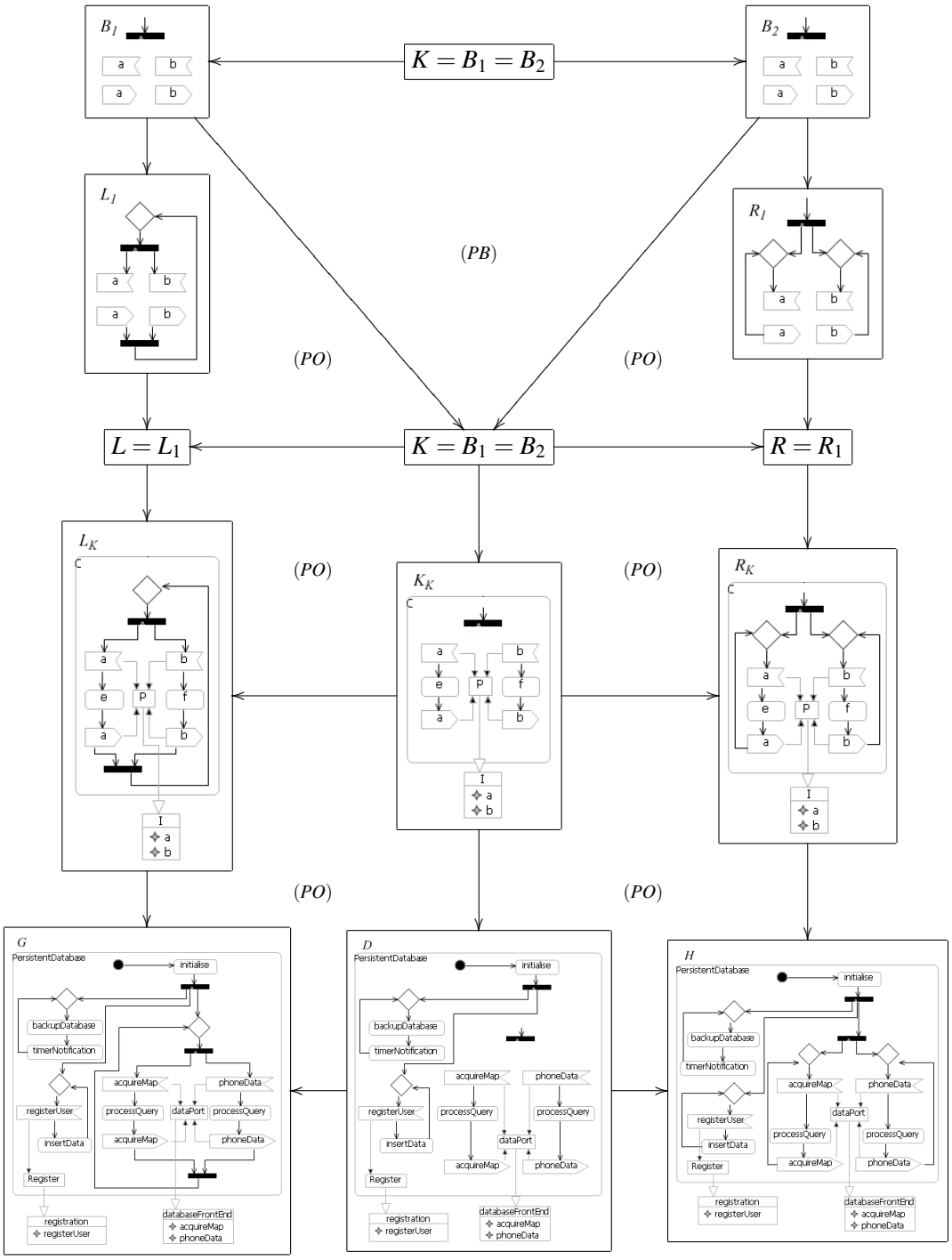
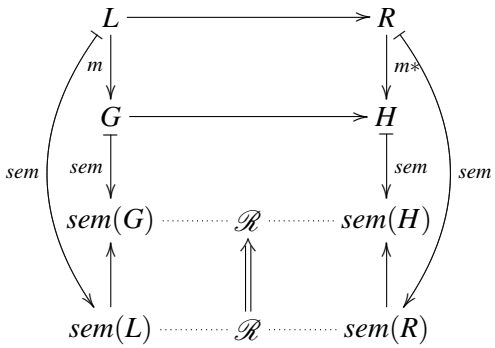


Figure 5: The construction of the minimal and extracted rule



The overall idea behind this is illustrated in the figure above [BHE08]. The original model (component, composite structure and activity diagrams) is given by graph G. The refactoring results in graph H by the application of rule $p : L \to R$ at match m. Applying the semantic mapping *sem* (itself described by a graph transformation from models to CSP) to the rule's left- and right-hand side, we obtain the CSP processes $sem(L)$ and $sem(R)$. Whenever the relation $sem(L)\ \mathscr{R}\ sem(R)$ (say $\mathscr{R}\ = \sqsubseteq$ is trace refinement, so all traces of the left processes are also traces of the right), we would like to be sure that also $sem(G)\ \mathscr{R}\ sem(H)$ (traces of $sem(G)$ are preserved in $sem(H)$).

The main assumption is the compositionality of the semantic mapping *sem*. Compositionality is similar to the compositionality property of denotational semantics. As for simple mathematical expressions, we assume that the the meaning of expression $2+5$ is determined by the meaning of 2, 5 and the semantics of the $+$ operator, i.e. $[[2+5]] = [[2]] \bigoplus [[5]]$.

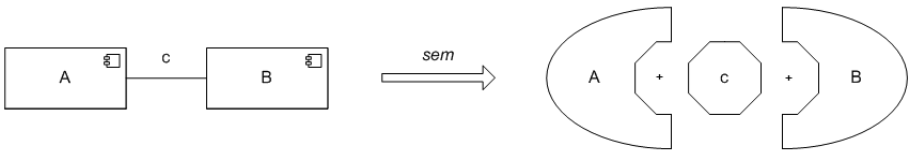


Figure 6: Compositional Semantic Mapping

In terms of model transformations, compositionality is presented in Figure 6. A system consisting of components A and B with a connector c is mapped to a semantic domain through transformation *sem*. The result is such a set of semantic expressions where $sem(A)$, $sem(B)$ and $sem(c)$ are distinguishable and their composition represents the semantics of the whole system.

Intuitively, the mapping must be closed under context, i.e., the semantics of a model L is embedded within the semantics of an extension G of L. Embedding of CSP processes is expressed by the notion of context, i.e., a CSP expression with a single occurrence of a distinguished free variable.

Definition 7 **(semantic domain)** A *semantic domain* is a triple $(D, \sqsubseteq, \mathscr{C})$ where D is a set, $\sqsubseteq$ is a partial preorder on D, $\mathscr{C}$ is a set of total functions $C[\,] : D \to D$, called contexts, such that $d \sqsubseteq e \implies C[d] \sqsubseteq C[e]$ ($\sqsubseteq$ is closed under contexts).

The equivalence relation $\equiv$ is the symmetric closure of $\sqsubseteq$.

Definition 8 **(compositionality)** A semantic mapping $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathscr{C})$ is *compositional* if, for any injective $m_0 : G_0 \to H_0$ and pushout (1) , there is a context E with $sem(H_0) \equiv E[sem(G_0)]$ and $sem(H_0') \equiv E[sem(G_0')]$

$$\begin{array}{ccc} G_0 & \longrightarrow & G_0' \\ {\scriptstyle m_0}\downarrow & (1) & \downarrow{\scriptstyle m_0'} \\ H_0 & \longrightarrow & H_0' \end{array}$$

Definition 8 applies particularly where G_0 is the left hand side of a rule and H_0 is the given graph of a transformation. In this case, the CSP expression generated from G_0 contains the one derived from H_0 up to *traces, failures or divergences* equivalence, while the context E is uniquely determined by $H_0 \setminus sem(G_0)$. The proof [BHE08] also relies on the fact that semantic relations $\mathscr{R}$ in CSP are closed under context.

Theorem 2 *Assume a compositional mapping* $sem : Graphs_{TG} \to (D, \sqsubseteq, \mathscr{C})$. *Then, for all transformations* $G \overset{p,m}{\Longrightarrow} H$ *via rule* $p : L \to R$ *with injective match* m, *it holds that* $sem(L) \sqsubseteq sem(R)$ *implies* $sem(G) \sqsubseteq sem(H)$.

6 Tool Support

This section discusses tool support that enables the developers to refactor system architecture, extract refactoring rules and verify them. The chain of tools that are used for rule extraction and verification is in Figure 7

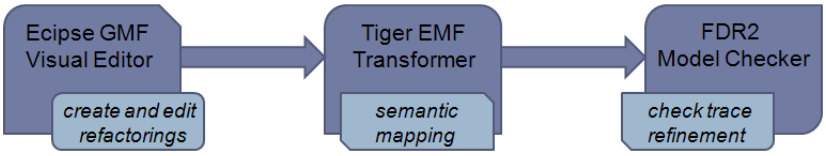


Figure 7: Block diagram of the tool chain

6.1 Visual Editor

A visual editor is necessary to perform the architectural refactoring on model level. The editor has been implemented using the Eclipse Graphical Modelling Framework (GMF) [GMF07].

As mentioned in Section 2, the components and their instances are situated in the same diagram, which is called a *combined structure diagram*. Such a diagram is shown in Figure 8. The round rectangle with label *PersistentDatabase* a type-level component, along with the *registration* and *databaseFrontEnd* interfaces. The behaviour is encapsulated within the component. The other rectangle with label *pdatabase* is a component instance. It contains two *interaction points* (i.e. port instances): *data1* and *register1*. Interface implementation is denoted by the usual inheritance and dependance connections: both interfaces are provided by the relevant ports.

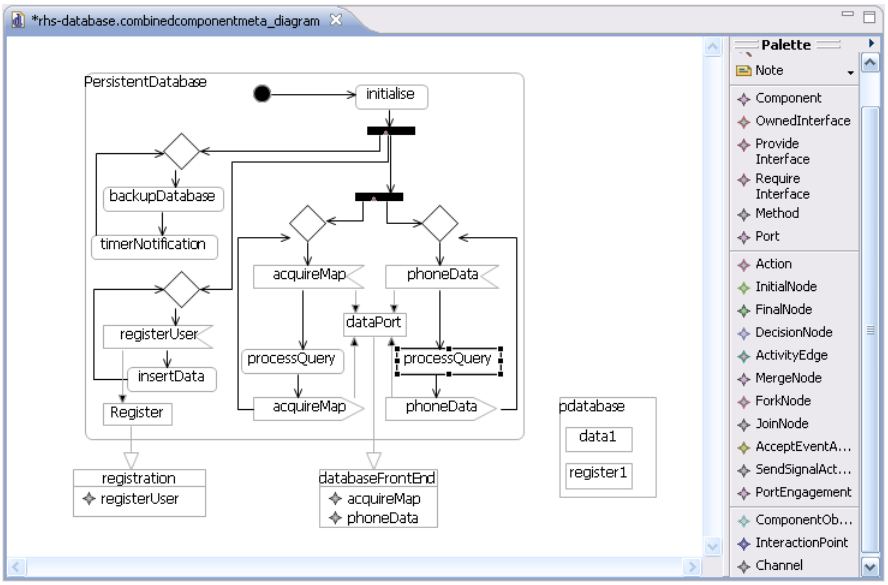


Figure 8: Basic Functionality of the Visual Editor

The metamodel of the combined structure [Bis08] is represented as an Eclipse Modeling Framework (EMF) model, which is essentially an attributed typed graph.

To help rule creation, every object in the diagram has an integer *id* value, which is 0 by default. This *id* expresses matches between the left- and right-hand side diagrams. Elements with the same *id* are matched.

6.2 Rule Extraction

It is desirable to have an automated method for minimal rule extraction. Although there are algorithms solving this problem [Var06], in our case the automated minimal rule extraction is future work.

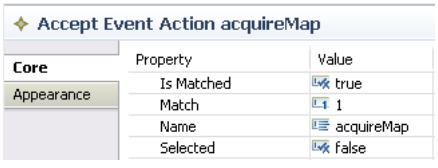


Figure 9: *Selected* attribute

Aside from editing, the visual editor enables the extraction of the refactoring rule. The attribute *Selected* is used for this purpose: it defines if the particular element is in the rule or not, regardless of its mach status. In Figure 9 the accept event action *acquireMap* is not included in the rule yet.

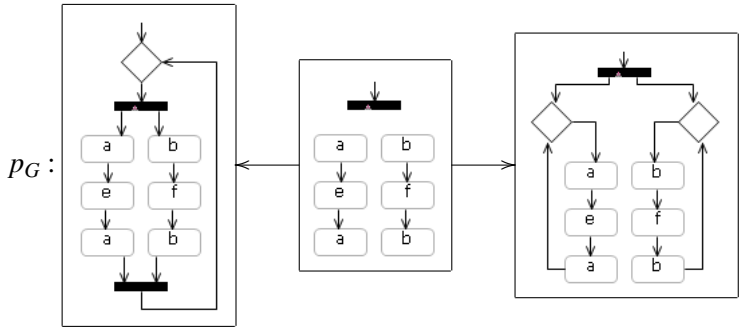


Figure 10: Generic Action in Rule

6.3 Context Inclusion

Consider the refactoring rule in Figure 10. It is similar to the context-included rule in Figure 4. Instead of communication actions, we used generic actions that match with *accept event actions* as well as *send signal actions*. It is obvious that this rule produces the same H when applied on G. The two rules are equivalent from the verification perspective.

It is possible to give guidelines that helps developers to either extract the rules themselves, or implement heuristics that suggest possible contexts. However, the included context, even in such a simple case is ambiguous. Semantical completeness may depend not only on the model itself, but also on the implementation of the semantic mapping.

In our case, the general guidelines for context inclusion are the following:

- if an *action* is present in G then all connected edges should be present as well
- if a *port* is present in G then all the connected *interfaces* should be present as well.
- if a *send signal action* or *accept event action* is present in G, then the engaged *port* should be present as well
- if a *component* is present in G, then its owned *initial node* should be present as well.

6.4 Mapping and Formal Verification

The transformation that maps the *combined structure diagram* to CSP is implemented using the Tiger EMF Transformer tool [Tig]. It consists of 45 rules organised in 4 major groups (type-level, owned behaviour, instance-level, renaming) as detailed in [Bis08]. The rules were designed using the EMT Visual Editor.

After the mapping to CSP, the expressions are checked for trace refinement with FDR2, a refinement checker for establishing properties of models expressed in CSP [FSEL05].

7 Conclusions

In this paper we addressed the the rule extraction problem for verification techniques based on compositional semantic mappings. A formal framework was presented that establishes the theo-

retical background of the rule creation process enabling the development of refactoring patterns. To apply the theoretical results in practice, a tool chain was presented that supports the developers to create and verify their refactoring rules.

Future work includes the continual development of the editing environment methods for possible automated rule extraction and further research on usable case studies.

Acknowledgements: This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

References

[BHE08] D. Bisztray, R. Heckel, H. Ehrig. Verification of Architectural Refactorings by Rule Extraction. In *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science 4961/2008, pp. 347–361. Springer Berlin / Heidelberg, 2008.

[Bis08] D. Bisztray. Verification of Architectural Refactoring Rules. Technical report, Department of Computer Science, University of Leicester, 2008. http://www.cs.le.ac.uk/people/dab24/refactoring-techrep.pdf.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In *Handbook of Graph Grammars*. Pp. 163–246. 1997.

[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.

[Ehr87] H. Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*. Pp. 3–14. Springer-Verlag, London, UK, 1987.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition edition, 1999.

[GMF07] Eclipse Graphical Modeling Framework. http://www.eclipse.org/gmf, 2007.

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

[Leh96] M. M. Lehman. Laws of Software Evolution Revisited. In *European Workshop on Software Process Technology*. Pp. 108–124. 1996. citeseer.ist.psu.edu/lehman96laws.html

[FSEL05] Formal Systems Europe Ltd. FDR2 User Manual. 2005. http://www.fsel.com/documentation/fdr2/html/index.html.

[Tig] Tiger Developer Team. Tiger EMF Transformer. http://www.tfs.cs.tu-berlin.de/emftrans.

[Var06] D. Varró. Model Transformation by Example. In *Proc. Model Driven Engineering Languages and Systems (MODELS 2006)*. LNCS 4199, pp. 410–424. Springer, Genova, Italy, 2006.

[WCG+06] M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, A. Schroeder. Semantic-Based Development of Service-Oriented Systems. In al. (ed.), *Proc. 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems(FORTE'06), Paris, France*. LNCS 4229, pp. 24–45. Springer-Verlag, 2006.
http://rap.dsi.unifi.it/sensoria/Sensoria_Forte.pdf