Bisimulation Verification for the DPO Approach with Borrowed Contexts

Guilherme Rangel[1], Barbara König[2] and Hartmut Ehrig[3]

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

[2] barbara_koenig@uni-due.de
Institut für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany

Abstract:

Bisimilarity is the most widespread notion of behavioral equivalence and hence algorithms for bisimulation checking are of fundamental importance for verifying that two systems are behaviorally equivalent (seen from the perspective of the environment). We investigate this problem in the context of behavioral equivalences of graphs and graph transformation systems, where the extension of the DPO approach to borrowed contexts provides us with a formal basis for reasoning about bisimilarity of graphs. In this paper we extend Hirschkoff's on-the-fly algorithm for bisimulation checking, enabling it to verify whether two graphs are bisimilar with respect to a given set of productions. We then apply this framework to refactoring problems and verify instances of a model transformation which describes the minimization of deterministic finite automata.

Keywords: Bisimulation, Graph Transformation, Refactoring, Automata

1 Introduction

Model transformation [MV05] concerns the automatic generation of models from other models according to a transformation definition, which describes how a model in the source language can be transformed into a model in the target language. Such transformations can take place between different models or, more specifically, inside one single model (refactoring). Software refactoring is a modern software development activity to cope with the internal modification of source code to improve system quality, without changing the observable behavior.

Graph transformation systems (GTS) are well-suited to model not only refactorings but also model transformation (see [MT04] for the correspondence between refactoring and GTS). A GTS specifies model transformation by defining graph transformation rules to translate one model into another. The general idea is to have a graph describing an

instance of the source model as a start graph and to apply graph productions until no further production can be applied and the resulting graph is an instance of the target model. Model transformations via GTS can be found in [EE06, EW06, VVE+06]. A crucial question that must be asked is whether a given refactoring (or model transformation) is behavior-preserving, which means that transforming one model into another model does not change the original external behavior. In practice, the proof of behavior-preserving transformations is not an easy task and therefore one normally relies on test suite executions and informal arguments in order to improve confidence that the behavior is preserved. In a recent paper Narayanan and Karsai [NK06] proposed a method for checking bisimilarity in model transformations using GTS which is similar to ours. The new contribution of our paper is to present an efficient bisimulation checking algorithm, which works on the fly for infinite state spaces, and to develop the theory in the very general framework of borrowed contexts [EK04].

In this paper we give a formal treatment of the question of behavior-preserving refactoring. We define model refactoring by graph productions in the Double Pushout Approach (DPO) [CMR+97], which is one of the standards for GTS. Our goal is to show that instances of one model are bisimilar to their refactored counterparts, which implies behavior preservation. We employ the extension of DPO to borrowed contexts [EK04], which provides the means to reason about bisimilarity. We also extend Hirschkoff's [Hir01] on-the-fly bisimulation checking algorithm to deal with our setting. A case study of refactoring is presented in terms of minimization of deterministic finite automata (DFA), where we can test if a given DFA is bisimilar to its minimal refactored version. Since the procedure to check bisimilarity by hand is quite tedious we have implemented a tool in Objective Caml [OCa] to support this activity.

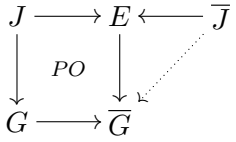
2 Graphs with Interfaces and Borrowed Contexts

In this section we recall the DPO approach to graph rewriting and its extension with borrowed contexts.

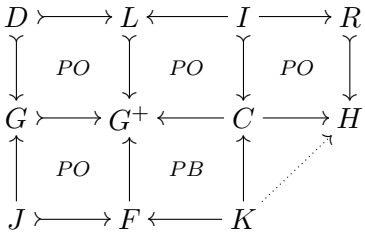
Definition 1 (**Graph and graph morphism**) A *graph* $G = (V, E, s, t, l_v, l_e)$ consists of a set V of nodes, a set E of edges, two functions $s, t\colon E \to V$ (source and target) and two labeling functions for nodes and edges $l_v\colon V \to \Omega_V$, $l_e\colon E \to \Omega_E$, where Ω_V and Ω_E are node and edge labels. A *graph morphism* $f\colon G_1 \to G_2$ is a pair of functions $f = (f_E\colon E_1 \to E_2, f_V\colon V_1 \to V_2)$, which is compatible with source, target and labeling functions of G_1 and G_2, i.e., $f_V \circ s_1 = s_2 \circ f_E$, $f_V \circ t_1 = t_2 \circ f_E$, $l_{e_2} \circ f_E = l_{e_1}$ and $l_{v_2} \circ f_V = l_{v_1}$.

In the standard DPO approach, graph productions rewrite graphs with no interaction with any other entity than the graph itself and the production. In the DPO with borrowed contexts [EK04] graphs have interfaces and may borrow missing parts of left-hand sides from the environment via the interface. This leads to open systems which take into account interaction with the outside world.

Definition 2 (**Graphs with interfaces and graph contexts**) A *graph G with interface* J is a morphism $J \to G$ and a context consists of two morphisms $J \to E \leftarrow \overline{J}$. The *embedding*[1] of a graph with interface $J \to G$ into a context $J \to E \leftarrow \overline{J}$ is a graph with interface $\overline{J} \to \overline{G}$ which is obtained by constructing $\overline{G}$ as the pushout of $J \to G$ and $J \to E$.



Definition 3 (**Rewriting with borrowed contexts**) Given a graph with interface $J \to G$ and a production $p\colon L \leftarrow I \to R$, we say that $J \to G$ reduces to $K \to H$ with transition label[2] $J \to F \leftarrow K$ if there are graphs D, G^+, C and additional morphisms such that the diagram below commutes and the squares are either pushouts (PO) or pullbacks (PB) with injective morphisms. In this case a *rewriting step with borrowed context* (BC for short) is feasible: $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$.



Consider the diagram above. The upper left-hand square merges L and the graph G to be rewritten according to a partial match $G \leftarrow D \to L$. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with a morphism $J \to F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting graph H, which can be obtained by "intersecting" the borrowed context F and the graph C via a pullback. Note that the two pushout complements that are needed in Definition 3, namely C and F, may not exist. In this case, the rewriting step is not feasible. Let us also remark that the arrows depicted as $\to$ in the diagram above can also be non-injective (see [SS05]).

Note that our notion of labels exactly coincides with labels derived by relative pushouts [LM00, SS05].

3 Bisimilarity

Here we show how to use transition labels in order to check bisimilarity between two graphs with interfaces. A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

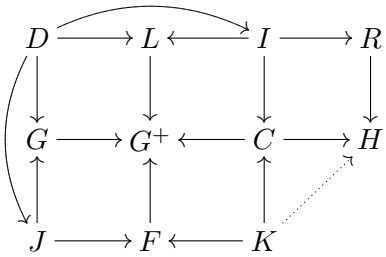
[1] The embedding is defined up to isomorphism since the pushout object is unique up to isomorphism.
[2] Transition labels, derived labels and labels are synonyms in this paper.

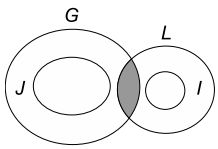
Definition 4 (**Bisimulation and Bisimilarity**) Let $\mathcal{P}$ be a set of productions and $\mathcal{R}$ a symmetric relation containing pairs of graphs with interfaces $(J \to G, J \to G')$. The relation $\mathcal{R}$ is called a *bisimulation* if, whenever we have $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$, then there exists a graph with interface $K \to H'$ and a transition $(J \to G') \xrightarrow{J \to F \leftarrow K} (K \to H')$ such that $(K \to H)\,\mathcal{R}\,(K \to H')$.

We write $(J \to G) \sim (J \to G')$ whenever there exists a bisimulation $\mathcal{R}$ that relates the two graphs with interface. The relation $\sim$ is called *bisimilarity*.

In the graph setting not all labels that can be derived from a graph and a set of productions are relevant for the bisimulation. We will distinguish two kinds of transition labels.

Definition 5 Let $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$ be a transition of $(J \to G)$. We say that the transition is *independent* whenever we can add two morphisms $D \to J$ and $D \to I$ to the diagram in Definition 3 such that the diagram below commutes, i.e., $D \to I \to L = D \to L$ and $D \to J \to G = D \to G$. We write $(J \to G) \xrightarrow{J \to F \leftarrow K}_d (K \to H)$ if the transition is not independent and we call it *dependent*.





An independent label has a borrowed context F that provides the entire left-hand side L for G and hence G does not contribute to the rewriting. (A trivial example is a label derived with $D = \emptyset$.) The figure above on the right schematically depicts this situation where the partial match occurs only in the overlap of the interfaces J and I leading to an independent label.

The bisimulation game for graphs mainly takes dependent labels into account. That is, if we modify Definition 4 in such a way that only dependent transitions $(J \to G) \xrightarrow{J \to F \leftarrow K}_d (K \to H)$ have to be simulated (either by a dependent or independent transition), then the resulting bisimilarity $\sim$ is unchanged (see [EK04]).

One of the main advantages of the borrowed contexts technique is that the derived bisimilarity is automatically a congruence, which means that whenever one graph with interface is bisimilar to another, one can exchange them in a larger graph without effect on the observable behavior. This is very useful for model refactoring since we can replace one part of the model by another bisimilar one.

Theorem 1 (**Bisimilarity is a Congruence** [EK04]) *The bisimilarity relation $\sim$ is a congruence, i.e., it is preserved by contextualization as given in Definition 2.*

Bisimulation proofs often yield infinite relations. Hence up-to techniques [San95] for bisimulation are useful to relieve the onerous task of bisimulation proofs by reducing the size of the relation needed to define a bisimulation. Bisimulation up-to is defined by replacing $(K \to H)\,\mathcal{R}\,(K \to H')$ by $(K \to H)\,\mathcal{F}(\mathcal{R})\,(K \to H')$ in Definition 4, where $\mathcal{F}$ is a function from relations to relations that defines the up-to technique (for details see [EK04]). We use for instance $\mathcal{F}^{iso}$ which generates all isomorphic copies of every pair in $\mathcal{R}$. A more powerful up-to technique is given by $\mathcal{F}^{context}$, which embeds all pairs into the same contexts (as in Definition 2), for all pairs and all compatible contexts.

The search of dependent labels among several partial matches might lead to cases where the pushout complement F or C (see Definition 3) does not exist and so the borrowed context step is not feasible. In [BGK06] a technique, based on initial pushouts, is defined to check if a partial match allows the existence of F and C.

Proposition 1 *[BGK06] Let $p\colon L \leftarrow I \to R$ be a production and $f\colon D \to L$ a monomorphism such that the diagram below on the left is the initial pushout of f. The pushout complement F of Definition 3 exists if and only if there is a monomorphism $D \to G$ and a morphism $J_D \to J$ such that the diagram on the right commutes.*

$$\begin{array}{ccc} J_D & \longrightarrow & F_D \\ {\scriptstyle g}\downarrow & IPO & \downarrow \\ D & \xrightarrow[f]{} & L \end{array} \qquad \begin{array}{ccc} J_D & \xrightarrow{g} & D \\ \downarrow & = & \downarrow \\ J & \longrightarrow & G \end{array}$$

Symmetrically one can check that the pushout complement C exists by taking the initial pushout over $D \to G$.

4 Partial Match Finding

Here we propose an algorithm that takes as input a graph with interface $J \to G$ and a set $\mathcal{P}$ of productions of the form $p\colon L \leftarrow I \to R$ to find all possible partial matches $G \leftarrow D \to L$ that will lead to dependent labels. We first need to introduce partial morphisms.

Definition 6 (Partial graph morphism) Let $G = (V, E, s, t, l_v, l_e)$ be a graph as in Definition 1. A *subgraph* S of G, written $S \subseteq G$, is a graph with $V^S \subseteq V^G$, $E^S \subseteq E^G$, $s^S = s^G|_{E^S}$, $t^S = t^G|_{E^S}$, $l_v^S = l_v^G|_{V^S}$ and $l_e^S = l_e^G|_{E^S}$. A *partial graph morphism* $f\colon G \rightharpoonup G'$ is a total graph morphism $f\colon dom(f) \to G'$ from a subgraph $dom(f) \subseteq G$ to G'.

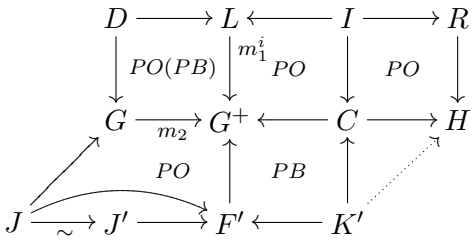
Given L (the left-hand side of a production) and G, we try to find partial matches which lead to a feasible BC step with a dependent label. We describe a procedure in 5 steps for one single production p, but it must be carried out for all productions of $\mathcal{P}$. Step **1** determines a subgraph L^{clean} of L, which is the largest subgraph of L containing only node and edge labels that also occur in G. The graph G^{clean} is defined analogously (with the roles of L and G exchanged). Step **2** creates all possible subgraphs L_i^{sub} $(i \in \mathbb{N})$ of L^{clean}. Step **3** finds all injective partial matches $pm_j\colon L_i^{sub} \rightharpoonup G^{clean}$

($j \in \mathbb{N}$). Step **4** splits each partial match pm_j as a span of total injective morphisms $L \leftarrow L^{clean} \leftarrow D_j \rightarrow G^{clean} \rightarrow G$. Step **5** stores $L \leftarrow D_j \rightarrow G$ as a partial match to derive a label if $L \leftarrow D_j \rightarrow G$ satisfies the conditions of Proposition 1 (for the existence of F and C) and will lead to a dependent label (Definition 5).

5 Matching Labels and Existence of Derivable Labels

The bisimulation game for graphs demands the comparison of labels. More specifically, two labels $\mu_i = J_i \rightarrow F_i \leftarrow K_i$ ($i = 1, 2$) are called isomorphic ($\mu_1 \cong \mu_2$) if they are isomorphic cospans. Remember that a dependent label can be answered by either a dependent or independent label. Since the algorithm in the previous section derives only dependent labels, we propose a way to check whether a dependent label for one graph is also derivable for the other graph. This if more efficient than deriving all independent labels for the other graph, which could be a lot, and checking whether they match.

Definition 7 (**Derivable Label**) Given a graph $J \rightarrow G$, a label $J' \rightarrow F' \leftarrow K'$ and a set $\mathcal{P}$ of productions, we say that $J' \rightarrow F' \leftarrow K'$ is *derivable* from $J \rightarrow G$ and $\mathcal{P}$ if it yields a feasible BC step, as in Definition 3.



This can be checked as follows: if there exists an isomorphism $J \xrightarrow{\sim} J'$ we obtain $J \rightarrow F'$ as the composition $J \xrightarrow{\sim} J' \rightarrow F'$ and in addition $G \rightarrow G^+ \leftarrow F'$ as a pushout of $G \leftarrow J \rightarrow F'$. For all productions $\mathcal{P}$ we find all possible total matches $m_1^i \colon L \rightarrow G^+$ ($i \in \mathbb{N}$). For each m_1^i and $m_2 \colon G \rightarrow G^+$, if m_1^i and m_2 are jointly surjective (i.e., $m_{1,V}^i(L_V) \cup m_{2,V}(G_V) = G_V^+$ and $m_{1,E}^i(L_E) \cup m_{2,E}(G_E) = G_E^+$) we can take $G \leftarrow D \rightarrow L$ as a pullback of $G \rightarrow G^+ \leftarrow L$ and thus obtain a pushout. We compute the pushout complement $G^+ \leftarrow C \leftarrow I$ of $G^+ \leftarrow L \leftarrow I$ and the pushout $C \rightarrow H \leftarrow R$ of $C \leftarrow I \rightarrow R$. We then check if there exists a morphism $K' \rightarrow C$ such that the rightmost square in the second row is a pullback and add the induced morphism $K' \rightarrow H$. If there exists a total match $m_1^i \colon L \rightarrow G^+$, which allows us to complete this diagram, we say that the label $J' \rightarrow F' \leftarrow K'$ is derivable from $J \rightarrow G$ and $\mathcal{P}$. Note that this is easier than partial match finding since we are only looking for total matches.

6 Algorithm for Bisimulation "On the Fly"

Classical methods for bisimulation checking (e.g., see [PT87]) take as input the full state spaces which are derived from the initial processes to be compared. Their drawback is

that the whole state space must first be computed and stored. Fernandez and Mounier defined in [FM91] a method for building the state space on the fly and checking bisimilarity based on depth-first search (DFS). Hirschkoff [Hir01] extended their work to not only allow breadth-first search (BFS), but also to deal with bisimulation up-to.

Hirschkoff's algorithm takes two states P and Q of labeled transition systems (LTSs) and checks their bisimilarity by analyzing their state space product, which consists of pairs of the form (P, Q) as states and transition labels μ between states indicating that both states are able to evolve along the same label μ, i.e., $(P, Q) \xrightarrow{\mu} (P', Q')$. The algorithm initially checks whether P and Q are immediately bisimilar (none of them has further labels leading to successor states) or non-bisimilar (one makes a step which the other is not able to mimic). If P and Q are not found immediately (non-)bisimilar, their state space product is expanded by adding their successors reached by a common label and so the bisimilarity of (P, Q) can be only known after the recursive analysis of all successors in the state space product. With this basic technique the LTSs in question must be finite. But in some cases the algorithm is able to perform finite proofs for states whose state space product is infinite using up-to techniques in order to handle infinite bisimulations as finite bisimulations up-to. Hirschkoff proved that the breadth-first version of the algorithm is computationally complete with respect to a given up-to technique $\mathcal{F}$, which means that the algorithm can check the bisimilarity of two states if and only if a finite bisimulation up to $\mathcal{F}$ relating the two states to be checked exists. Hirschkoff used his algorithm to check bisimilarity of polyadic π-calculus [Mil93] processes.

We extend Hirschkoff's algorithm to check bisimilarity between graphs with interfaces with respect to a given set of graph productions. We also did minor efficiency improvements and added extra details to the algorithm, trying to make clear aspects that were not easy to understand in the original version.

Remember that in order to calculate all dependent labels which originate from a given graph with interface and a set $\mathcal{P}$ of productions we employ the algorithm defined in Section 4 to find partial matches between the left-hand side of the rules of $\mathcal{P}$ and the graph with interface. For every partial match we then use Definition 3 to complete the whole borrowed context diagram, which gives us the dependent label and the resulting graph with interface. The matching of labels is specified in Definition 7.

In the setting of graphs, the bisimulation checking algorithm explores the state space product (defined below) of two graphs to be compared. We use here some shortcuts: graphs with interfaces $J \to G$ are represented as P and Q, and labels $J \to F \leftarrow K$ as μ.

Definition 8 (State Space Product) The *state space product* of two graphs P_0 and Q_0 is the transition system generated from the initial state (P_0, Q_0) using the following inference rules:

$$\text{dep1}: \quad \frac{P \xrightarrow{\mu}_d P' \quad Q \xrightarrow{\mu'} Q'}{(P,Q) \xrightarrow{\mu} (P',Q')} \qquad \text{dep2}: \quad \frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\mu'}_d Q'}{(P,Q) \xrightarrow{\mu} (P',Q')} \qquad \mu \cong \mu'$$

The successors of (P, Q) are all (P', Q') such that P' and Q' respectively correspond to evolutions of P and Q along an isomorphic label μ. The rules *dep1* and *dep2* cover the situation when one dependent label (indicated with $\to_d$ above) is answered (i.e. matched)

by either a dependent or independent isomorphic label. If one graph can not answer, we say that the pair fails to evolve, i.e., we can infer immediately that P and Q are not bisimilar.

Definition 9 (**Failure**) Given two graphs P and Q we say that the pair (P, Q) *fails* to evolve whenever it holds:
$(P \xrightarrow{\mu}_d P' \wedge \nexists Q' \colon Q \xrightarrow{\mu'} Q' \text{ s.t. } \mu \cong \mu') \vee (Q \xrightarrow{\mu}_d Q' \wedge \nexists P' \colon P \xrightarrow{\mu'} P' \text{ s.t. } \mu \cong \mu')$.

The data structures used by the algorithm are: a structure **S** containing pairs of states that still have to be inspected; a *Table* storing information about each pair of graphs under inspection and three sets V, W and R, containing pairs of graphs that are respectively supposed to be bisimilar, known to be non-bisimilar and known to be bisimilar. By accessing **S** as stack (resp. queue) the algorithm performs a depth-first (resp. breadth-first) search on the state space product.

The main procedure is **bisimulation_check**, which calls: **succeeds**, **fails** and **propagate**. The procedure **bisimulation_check** first checks with **succeeds** whether (P, Q) is immediately bisimilar (e.g. none of them is able to derive any further (dependent) label). If (P, Q) is not immediately bisimilar, **fails** checks if the pair fails to evolve or if it is already known as non-bisimilar (i.e., it is in W). If it fails we insert it into W and use **propagate** to update the information about the state space product in *Table* with this new result, which can possibly lead to the discovery of new (non-)bisimilar graphs. When we find that a new pair evolves to other pairs, we assume that it is bisimilar (insert it into V) and only after the analysis of all its successors (where the notion of successor is specified in Definition 8) we are able to decide whether the pair is really bisimilar (as we assumed) or not. If by processing **S** we find a pair that is in V (supposed to be bisimilar) we move it to R and update *Table* using **propagate**. When all pairs have already been analyzed ($\mathbf{S} = \emptyset$) the algorithm can determine the bisimilarity of (P, Q).

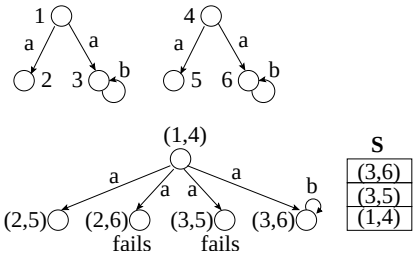


Table			
(P, Q)	*successors*	m	*fails*
$(1, 4)$	(2,5) (2,6) (3,5) (3,6) ● ● ○ ○	2 3 5 6 ☒ ☐ ☒ ☐	*false*
$(3, 5)$			*true*
$(3, 6)$	(3,6) ○	3 6 ☐ ☐	*false*

Above, one can see two small transition systems, their respective state space product, the states under investigation in **S** and their current information in *Table*. The states 1–6 represent graphs and a, b are labels. Consider the pair $(1, 4)$ in *Table*. The entry *successors* shows the successors of $(1, 4)$ in the state space product together with a boolean value *true* (●) or *false* (○), indicating which pairs of successors have already been analyzed. The entry m lists the successor states (e.g. 3 and 6 with *false* [☐], 2 and 5 with *true* [☒]), indicating which state has found a bisimilar partner. Whenever a pair of successors has been analyzed, if it turns out to be bisimilar both m-fields of the pair are set to *true* (☒). Successors $(2, 5)$ and $(2, 6)$ have been explored and only

$(2,5)$ is bisimilar. If all successors of (P,Q) have been analyzed (all are set to $\bullet$) and all fields of m are set to *true* ($\boxtimes$) then (P,Q) is bisimilar. If there is in m at least one graph with *false* ($\square$) then (P,Q) is not bisimilar. The entry *fails* indicates if a pair fails to evolve (according to Definition 9). A non-bisimilar pair has always at least one successor leading to a failure, but it is worth observing that a bisimilar pair might also have successors leading to a failure. In the example even though $(2,6)$ and $(3,5)$ fail, it is clear that $(1,4)$ is bisimilar.

```
bisimulation_check(P,Q) :=
  W := ∅;
  (*) R := ∅; V := ∅;
   insert (P,Q) into S and Table; status := true;
  while S ≠ ∅ do
     take (P0,Q0) from S;
     if succeeds((P0,Q0))
        then insert (P0,Q0) into R;
             propagate((P0,Q0), true);
     else if fails ((P0,Q0))
             then insert (P0,Q0) into W;
                  propagate((P0,Q0), false);
     else if [(P0,Q0) ∈ V]
             then move (P0,Q0) from V to R;
                  propagate((P0,Q0), true);
     else if [(P0,Q0) ∈ R]
             then propagate((P0,Q0), true);
     else {pair (P0,Q0) is new}
             insert (P0,Q0) into V;
             {(P0,Q0) -μ-> successor(P0,Q0)}
             insert successors of (P0,Q0) into S;
             for each successor(P0,Q0) do
               if successor(P0,Q0) ∉ Table
                 ∧ successor(P0,Q0) ∉ W ∪ R ∪ V
                 then insert it into Table;
             end for
  end while
if (P,Q) ∉ R
  then return false
  else if status then return true else loop back to (*)
```

```
succeeds(P,Q) := Table(P,Q).successors = ∅
    ∧ Table(P,Q).fails = false

fails(P,Q):=
  if (P,Q) ∈ Table
     then Table(P,Q).fails ∨ (P,Q) ∈ W
     else (P,Q) ∈ W

propagate((P,Q), success) :=
  if (P,Q) ∈ R ∧ success = false
       then status := false;
  if (P,Q) ∈ Table
     ∧ Table(P,Q).successors is complete[3]
       then remove (P,Q) from Table;
  for each (Pf,Qf) ∈ Table with (Pf,Qf) -μ-> (P,Q) do
       Table(Pf,Qf).successors(P,Q) := true;
       if success
          then Table(Pf,Qf).m(P) := true;
               Table(Pf,Qf).m(Q) := true;
       if Table(Pf,Qf).successors is complete
         then
           if ∃j = false ∈ Table(Pf,Qf).m
            then
               insert (Pf,Qf) into W;
               propagate ((Pf,Qf), false);
            else
               if (Pf,Qf) ∈ V
                  then take it from V to R;
               propagate ((Pf,Qf), true);
  end for
```

A new pair (P,Q) is inserted into *Table* as follows. Using Definition 8 and Definition 9 we can determine if (P,Q) fails to evolve and if it has successors. We insert (P,Q) into *Table* with the following data in case of failure: $successors = \emptyset$, $m = \emptyset$ and $fails = true$. If the pair does not fail we fill *successors* with the successors of (P,Q), m with the states of the successors of (P,Q) and all boolean values are set to *false*.

The procedure **propagate** is in charge of updating the information in *Table* concerning the state space product analysis. Every time we rediscover a new pair we decide that it is bisimilar (i.e., it is moved to R). If during the exploration of the state space we find that this very pair is non-bisimilar we set the variable *status* to *false*, which means that the result of the current run of the algorithm is not reliable. In this case we restart it retaining the information in W about states which are already known to be non-bisimilar.

The procedure **propagate** keeps in *Table* only the pairs under analysis, removing the

[3] This means that either all successors of (P,Q) have already been analyzed ($\bullet$) or $successors = \emptyset$.

ones that have already been analyzed. Observe that when one pair is propagated, the predecessors of this pair should also be informed of the new results. When a pair is propagated with *true* or *false* the algorithm always sets this pair as analyzed (*true* [•]) in the list of *successors* of each of its predecessors that are still under analysis (in *Table*). Only if the pair is propagated with *true* its respective graphs in m of its predecessors are set to *true* (☒). When the last successor of a given pair has been analyzed we can verify whether our initial hypothesis about its bisimilarity is still true. If the pair has at least one graph in m set as *false* it is non-bisimilar (our hypothesis was wrong) and we propagate this result. Otherwise we conclude that our hypothesis was correct and propagate this information.

If the algorithm has to handle bisimulation up-to we have only to replace in **bisimulation_check** $\boxed{(P_0, Q_0) \in V}$ by $\boxed{(P_0, Q_0) \in \mathcal{F}(V)}$ and $\boxed{(P_0, Q_0) \in R}$ by $\boxed{(P_0, Q_0) \in \mathcal{F}(R)}$, where $\mathcal{F}$ describes the up-to technique. For the refactoring proposed in this paper (see Section 8) we use the up-to context technique ($\mathcal{F}^{context}$) [EK04] informally described in Section 3.

Our version of the bisimulation checking algorithm is very similar to Hirschkoff's. Our main contribution to the algorithm is the full specification of how *Table* is used to store and process the state space product investigation. A small efficiency improvement can be seen in the **for each** statement of **bisimulation_check**, where we added extra conditions in order to avoid reanalyzing pairs already investigated. Furthermore we checked that the algorithm also works in our setting of borrowed contexts, taking into account especially the issue of independent and dependent labels.

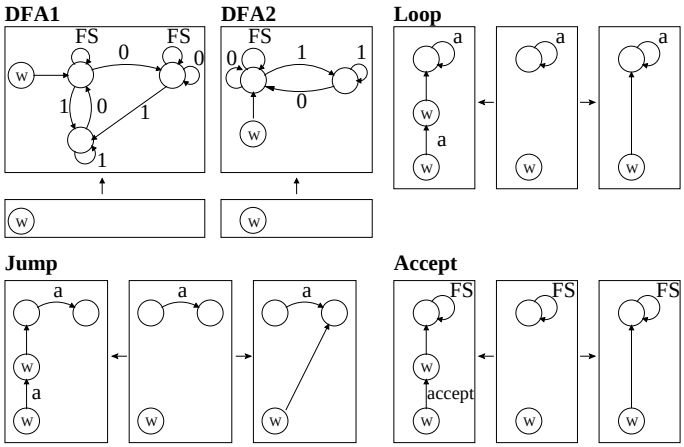
7 Tool Support for DPO with Borrowed Contexts

The derivation of labels and the bisimulation proof demand a great amount of time even for small examples and when done by hand they are particularly susceptible to errors. To overcome this we have implemented a tool in Objective Caml (OCaml), which is a functional language very appropriate for rapid prototyping. The tool uses directed labeled graphs and when we want to check the bisimilarity of two graphs, we specify a set of graph productions and also the graphs with interfaces to be checked. We have already implemented graphs with interfaces, graph productions, and procedures for label derivation and matching. OCaml is mainly textual, but for the sake of visualization, our graphs, rules and derived labels can be visualized with the package Graphviz [gra]. Our next goal is to implement the described algorithm for bisimulation checking.

8 Refactoring Deterministic Finite Automata

We will now come back to our original motivation: showing that refactoring preserves behavior. Let us first explain how the current theory of DPO with borrowed contexts could be used to prove that a refactoring process is behavior-preserving. Consider a model M, whose operational semantics is given by a set $OpSem^M$ of graph productions and a refactoring for this model M as a set $Refactoring^M$ of productions of the form

$L \leftarrow I \rightarrow R$. If we can prove for each rule of $Refactoring^M$ that we have the bisimilarity $(I \rightarrow L) \sim (I \rightarrow R)$ with respect to $OpSem^M$, this means that each rule does not change the behavior of the model under refactoring. Since bisimilarity is a congruence we can compose $(I \rightarrow L)$ and $(I \rightarrow R)$ with identical contexts and the respective compositions remain bisimilar. That is, for all instances of M, $Refactoring^M$ preserves the original behavior. This approach can be currently used only for refactoring rules without features such as negative application conditions and layers, which are often necessary to model refactorings.



Each DFA is described as a graph, where nodes are the states and directed labeled edges are the transitions (see DFA1 and DFA2 above). A loop labeled FS marks a state as final. An interface node labeled W has an edge pointing to the current state and this edge points initially to the start state. Note that the node W is the only possibility to establish interaction with the environment. The node W receives a letter (e.g. 'a') from the environment in form of an a-edge connecting W-nodes. Then, according to the operational semantics (given by the rules **Jump**, **Loop** and **Accept**) the DFA may change its state. Whenever an *accept*-edge between W-nodes is consumed by a DFA, this means that the string previously processed was accepted.

Below we define graph productions to minimize DFAs by eliminating equivalent states. The idea of the algorithm is to identify the distinguishable states, followed by the merging of equivalent states. Note that to the left of each rule we depict the negative application conditions. The algorithm is defined by several graph productions spread over three layers, where each layer applies its rules as long as possible before the rules of the next layer can be used. In practical terms, the transformation begins with rules of layer 0. If no more rule of layer 0 is applicable, the rules of layer 1 come into play. The rules in **layer 0** (see Figure 1) examine the transitions labels for every two states and determine if they are distinguishable. The rule in **layer 1** merges equivalent nodes, i.e., nodes without a *dist* edge between them. Finally, the rules in **layer 2** remove all *dist* edges and redundant transitions between two states.

Observe that the minimization algorithm demands rules spread over layers and negative application conditions and so we are not able to prove that all refactorings via

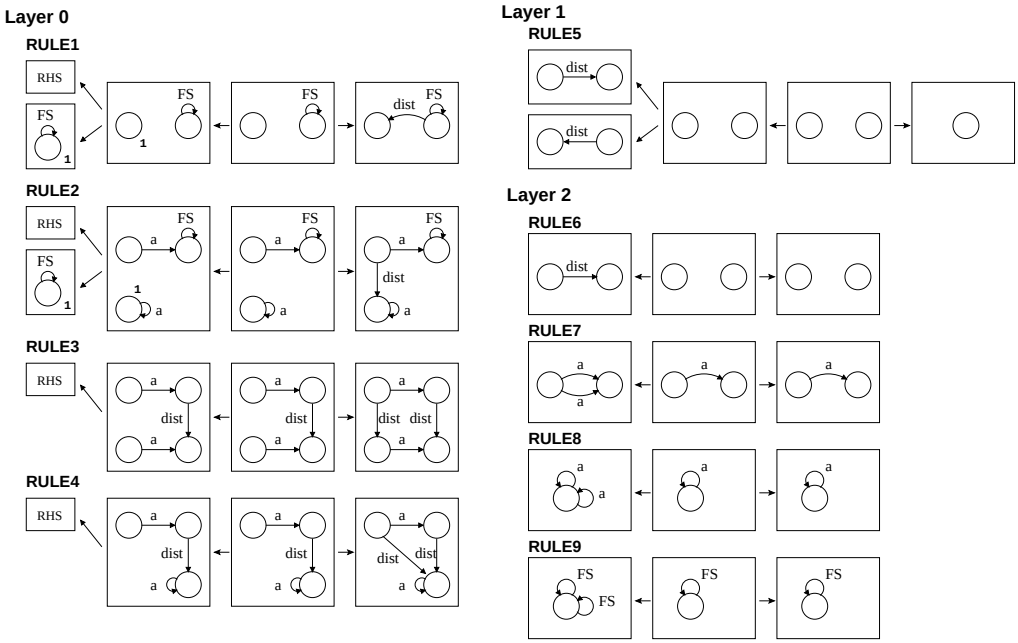


Figure 1: Productions for DFA minimization

these rules are behavior-preserving. For this reason our goal in this paper is to check that a given DFA and its minimal refactored version are bisimilar. Note that for DFAs borrowed context bisimilarity coincides with language equivalence. Furthermore in our setting bisimilarity on automata seen as transition systems corresponds to the bisimilarity that we obtain via the borrowed context technique.

Consider DFA1 and DFA2 previously depicted. By applying the minimization algorithm on DFA1 we obtain DFA2 as its minimal version. We then call the procedure **bisimulation_check** and verify that DFA1 and DFA2 are indeed bisimilar. In Figure 2 we show the state space product for this example, where the omitted interfaces of the graphs in the tuples contain only one node labeled W. Note that the state space product does not contain independent labels (which exist).

9 Conclusions and Future Work

We have shown how to use the DPO approach with borrowed contexts to automatically check the bisimilarity of systems specified in terms of graphs. Furthermore we suggested as a case study for refactoring the minimization of DFAs.

Our plan is to extend this work in such a way that whenever we define a refactoring as graph productions we should also be able to prove that all instances prior to refactoring are bisimilar to their refactored counterparts. The first step to be made to accomplish such an objective is the extension of the current theory of DPO with BC to handle rules with negative application conditions and layers, which are often used in refactorings.

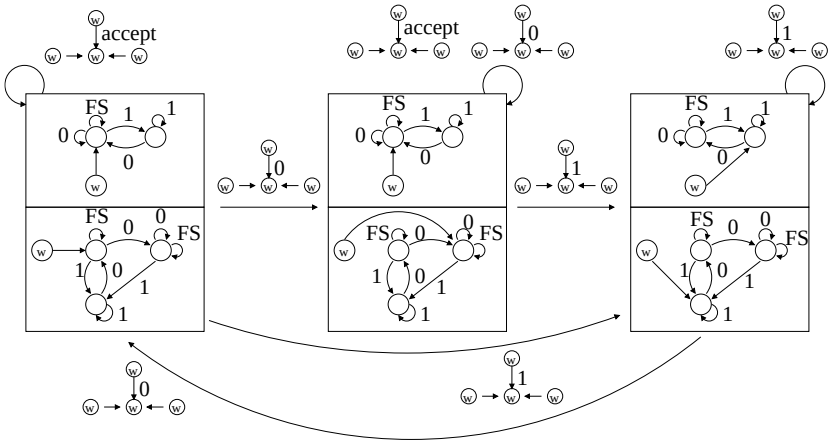


Figure 2: State space product for DFA1 and DFA2

Acknowledgements: We are grateful to Gabriele Taentzer and Paolo Baldan for interesting discussions on using DPO with borrowed contexts in the context of refactoring.

Bibliography

[BGK06] F. Bonchi, F. Gadducci, B. König. Process Bisimulation *via* a Graphical Encoding. In Corradini et al. (eds.), *Proc. of ICGT'06*. LNCS 4178, pp. 168–183. Springer, 2006.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Loewe. Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*. Pp. 163–246. World Scientific, 1997.

[EE06] H. Ehrig, K. Ehrig. Overview of Formal Concepts for Model Transformations Based on Typed Attributed Graph Transformation. In *Proc. of GraMoT '05*. ENTCS 152, pp. 3–22. Elsevier Science, 2006.

[EK04] H. Ehrig, B. König. Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting. In Walukiewicz (ed.), *Proc. of FoSSaCS '04*. LNCS 2987, pp. 151–166. Springer, 2004.

[EW06] K. Ehrig, J. Winkelmann. Model Transformation From VisualOCL to OCL Using Graph Transformation. In *Proc. of GraMoT '05*. ENTCS 152, pp. 23–37. 2006.

[FM91] J.-C. Fernandez, L. Mounier. On the Fly Verification of Behavioural Equivalences and Preorders. In *Proc. of CAV'91*. LNCS 757, pp. 181–191. Springer-Verlag, 1991.

[gra] Graphviz - Graph Visualization Software. Internet. http://www.graphviz.org.

[Hir01] D. Hirschkoff. Bisimulation Verification Using the Up to Techniques. *International Journal on Software Tools for Technology Transfer* 3(3):271–285, Aug. 2001.

[LM00] J. J. Leifer, R. Milner. Deriving Bisimulation Congruences for Reactive Systems. In *Proc. of CONCUR '00*. Volume 1877, pp. 243–258. Springer-Verlag, London, UK, 2000.

[Mil93] R. Milner. The polyadic π-Calculus: a tutorial. In Hamer et al. (eds.), *Logic and Algebra of Specification*. Springer-Verlag, Heidelberg, 1993.

[MT04] T. Mens, T. Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30(2):126–139, 2004.

[MV05] T. Mens, P. Van Gorp. A Taxonomy of Model Transformation. In *Proc. of GraMoT '05*. Volume 152, pp. 125–142. 2005.

[NK06] A. Narayanan, G. Karsai. Towards Verifying Model Transformations. In Bruni and Varró (eds.), *Proc. of GT-VMT '06*. ENTCS, pp. 185–194. Vienna, 2006.

[OCa] Objective Caml. http://caml.inria.fr/ocaml/.

[PT87] R. Paige, R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal on Computing* 16(6):973–989, 1987.

[San95] D. Sangiorgi. On the Proof Method for Bisimulation. In Wiedermann and Hájek (eds.), *Proc. of MFCS '95*. LNCS 969, pp. 479–488. Springer, 1995.

[SS05] V. Sassone, P. Sobociński. Reactive systems over cospans. In *Proc. of LICS '05*. Pp. 311–320. IEEE, 2005.

[VVE+06] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In Corradini et al. (eds.), *Proc. of ICGT '06*. LNCS 4178, pp. 260–274. Springer, Natal, Brazil, 2006.