Proceedings Paper:
Soeldner, Robert and Plump, Detlef orcid.org/0000-0002-1148-822X (2022) Towards Mechanised Proofs in Double-Pushout Graph Transformation. In: Proceedings of the Thirteenth International Workshop on Graph Computation Models. International Workshop on Graph Computation Models, 06 Jul 2022 Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, FRA, pp. 59-75.

https://doi.org/10.4204/EPTCS.374.6

Towards Mechanised Proofs in Double-Pushout Graph Transformation

Robert Söldner Detlef Plump
Department of Computer Science, University of York, York, UK
{rs2040,detlef.plump}@york.ac.uk

We formalise the basics of the double-pushout approach to graph transformation in the proof assistant Isabelle/HOL and provide associated machine-checked proofs. Specifically, we formalise graphs, graph morphisms and rules, and a definition of direct derivations based on deletion and gluing. We then formalise graph pushouts and prove with Isabelle's help that both deletions and gluings are pushouts. We also prove that pushouts are unique up to isomorphism. The formalisation comprises around 2000 lines of source text. Our motivation is to pave the way for rigorous, machine-checked proofs in the theory of the double-pushout approach, and to lay the foundations for verifying graph transformation systems and rule-based graph programs by interactive theorem proving.

1 Introduction

Software faults may lead to unexpected system's behaviour with a significant loss of goods or even personal harm. Documented examples of system failures range from medical devices [15] over space launch vehicles [7] to hardware design [13]. To prevent software faults, formal methods such as static analysis or program verification continue to attract a considerable amount of research.

Computing by rule-based graph transformation provides an intuitive and visual approach to specification and programming. Here, the main formal concepts for ensuring correctness are model checking [21, 25, 2, 20, 10] and proof-based verification [12, 14, 19, 24, 6, 23, 4, 28]. One of the oldest and most established approaches to graph transformation is the double-pushout (DPO) approach, where rule applications are defined by a pair of pushouts in the category of graphs [9]. Formal proofs in the DPO approach come in two flavours, they either establish results in the DPO theory (such as the commutativity of independent rule applications) or they show the correctness of concrete graph transformation systems and graph programs.

While mainstream formal methods increasingly employ proof assistants such as Coq [3] or Isabelle [16] to obtain rigorous, machine-checked proofs, to the best of our knowledge such tools have not yet been used in the area of DPO graph transformation. In this paper, we report on first steps towards the formalisation of the DPO theory in the Isabelle proof assistant. Specifically, we focus on linear rules with injective matching and show how to formalise (labelled, directed) graphs, morphisms, and rules. (Note that injective matching is more expressive than unrestricted matching because each rule can be replaced by the set of its quotient rules, and selected quotients can be omitted [11]). We give an operational definition of direct derivations based on deletion and gluing. We then formalise graph pushouts and prove with Isabelle's help that both deletion and gluing are pushouts. We also prove that pushouts are unique up to isomorphism.

We stress that we do not intend to formalise an abstract theoretical framework such as adhesive categories [9], nor do we aim at covering all kinds of graphs that one can find in the DPO literature such as infinite graphs, hypergraphs, typed graphs, etc. Rather, we are interested in concrete constructions on graphs

R. Heckel and C.M. Poskitt (Eds.): 13th International Workshop on Graph Computation Models (GCM 2022)
EPTCS 374, 2022, pp. 59–75, doi:10.4204/EPTCS.374.6

such as deletion and gluing, and how they relate to the double-pushout formulation. Our long-term goal is to provide interactive and automatic proof support for formal reasoning on programs in a graph transformation language such as GP 2 [5]. The underlying formalisation in Isabelle will inevitably have to deal with the concrete graphs, labels, rules, etc., which are the ingredients of such programs.

To summarize, this paper makes the following contributions:

- We formalise in Isabelle the basics of the DPO approach with injective matching.
- We prove that the operational construction of direct derivations by deletion and gluing gives rise to a double-pushout diagram.
- We prove that graph pushouts are unique up to isomorphism.

We believe that this is the first formalisation of DPO-based graph transformation in a theorem prover. The formalisation and proofs were developed using the Isabelle 2021 proof assistant. The entire formalisation comprises around 2000 lines of source text and can be accessed from GitHub[1].

This paper is a revised version of [22]. Here, we generalise our formalisation to support gluing and deletion with injective morphisms. Additionally, we follow Noschinski's [17] approach by using dedicated *record* types (for graphs and morphisms) and Isabelle's *locale* mechanism.

The rest of the paper is structured as follows: Section 2 briefly reviews the theoretical background required in this research. Section 3 will provide selected examples of our formalisation using the proof assistant Isabelle. Finally, in Section 4, the paper is summarised and future work is stated.

2 Graphs, Rules and Derivations

This section reviews basic terminology and results regarding graphs, rules, and derivations in the double-pushout approach with injective matching; see for example [9, 11]. In Section 3, we formalise these definitions and results in Isabelle.

Definition 1 (Label alphabet)**.** A *label alphabet* $\mathscr{L} = (\mathscr{L}_V, \mathscr{L}_E)$ consists of a set $\mathscr{L}_V$ of node labels and a set $\mathscr{L}_E$ of edge labels. □

We define directed and labelled graphs and allow parallel edges and loops. We do not consider variables as labels.

Definition 2 (Graph)**.** A *graph* $G = (V, E, s, t, l, m)$ over the alphabet $\mathscr{L}$ is a system where V is the finite set of nodes, E is the finite set of edges, $s, t \colon E \to V$ functions assigning the source and target to each edge, $l \colon V \to \mathscr{L}_V$ and $m \colon E \to \mathscr{L}_E$ are functions assigning a label to each node and edge. □

Next we review graph morphisms which are structure-preserving mappings between graphs. We describe our Isabelle formalisation in Subsection 3.1.

Definition 3 (Graph morphism)**.** A *graph morphism* $f \colon G \to H$ is a pair of mappings $f = (f_V \colon V_G \to V_H, f_E \colon E_G \to E_H)$, such that for all $e \in E_G$ and $v \in V_G$:

1. $f_V(s_G(e)) = s_H(f_E(e))$ (sources are preserved)
2. $f_V(t_G(e)) = t_H(f_E(e))$ (targets are preserved)
3. $l_G(v) = l_H(f_V(v))$ (node labels are preserved)

[1] https://github.com/UoYCS-plasma/DPO-Formalisation

4. $m_G(e) = m_H(f_E(e))$ (edge labels are preserved) □

We also define some special forms of morphisms.

Definition 4 (Special morphisms and isomorphic graphs)**.** A morphism f is *injective* (*surjective*, *bijective*) if f_V and f_E are injective (surjective, bijective). Morphism f is an *inclusion* if for all $v \in V_G$ and $e \in V_E$, $f_V(v) = v$ and $f_E(e) = e$. A bijective morphism is an *isomorphism*. In this case, G and H are *isomorphic*, which is denoted by $G \cong H$. □

The composition of two morphisms yields a well-defined morphism, which we prove in Subsection 3.1.

Definition 5 (Morphism composition)**.** Let $f\colon F \to G$ and $g\colon G \to H$ be graph morphisms. The *morphism composition* $g \circ f\colon F \to H$ is defined by $g \circ f = (g_V \circ f_V, g_E \circ f_E)$. □

In DPO-based graph transformation, rules are the atomic units of computation. We describe the formalisation of rules in Subsection 3.5.

Definition 6 (Rule)**.** A *rule* $(L \leftarrow K \rightarrow R)$ consists of graphs L, K and R over $\mathscr{L}$ together with inclusions $K \to L$ and $K \to R$. □

The addition of graph components along a common subgraph is called *gluing*. We present our Isabelle formalisation in Subsection 3.3. The gluing construction below uses the disjoint union of sets A and B defined by $A + B = (A \times \{1\}) \cup (B \times \{2\})$. It comes with injective functions $i_A\colon A \to A + B$ and $i_B\colon B \to A + B$ such that $i_A(A) \cup i_B(B) = A + B$ and $i_A \cap i_B = \emptyset$.

To keep the rest of this section readable, we tacitly assume that the injections i_A and i_B are inclusions. Only in section 3 we will be dealing explicitly with the injections. We prove the correspondence between the gluing construction and pushouts in Subsection 3.3.

Lemma 1 (Gluing [8])**.** *Let $b\colon K \to R$ and $d\colon K \to D$ be injective graph morphisms. Then the following defines a graph H (see Fig. 1a), the gluing of D and R according to d:*

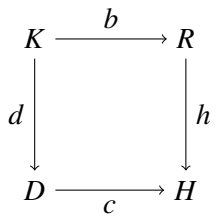
1. $V_H = V_D + (V_R - b_V(V_K))$
2. $E_H = E_D + (E_R - b_E(E_K))$
3. $s_H(e) = \begin{cases} s_D(e) & \text{if } e \in E_D \\ d_V(b_V^{-1}(s_R(e))) & \text{if } e \in E_R - b_E(E_K) \text{ and } s_R(e) \in b_V(V_K) \\ s_R(e) & \text{otherwise} \end{cases}$
4. t_H *analogous to* s_H
5. $l_H = \begin{cases} l_D(v) & \text{if } v \in V_D \\ l_R(v) & \text{otherwise} \end{cases}$
6. m_H *analogous to* l_H

*Moreover, the morphism $D \to H$ is an inclusion and the injective morphism h is defined for all items x in R by $h(x) =$ **if** $x \in R - b(K)$ **then** x **else** $d(x)$.*

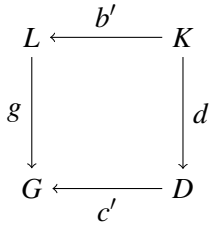
The dangling condition ensures that deletion results in a well-defined graph.

Definition 7 (Dangling condition)**.** Let $b'\colon K \to L$ be an injective graph morphism. An injective graph morphism $g\colon L \to G$ satisfies the *dangling condition* if no edge in $E_G - g_E(E_L)$ is incident to a node in $g_V(V_L - b'_V(V_K))$. □

The following *deletion* of graph components is formalised in Subsection 3.4.



(a) Gluing diagram



(b) Deletion diagram

Figure 1: Gluing and deletion diagram

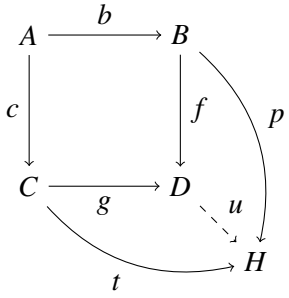


Figure 2: Pushout diagram

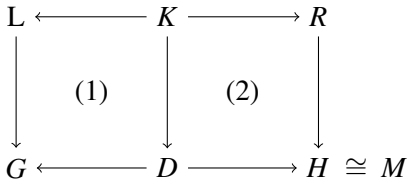


Figure 3: Direct derivation

Lemma 2 (Deletion [8])**.** *Let $b' \colon K \to L$ and $g \colon L \to G$ injective graph morphisms and let g satisfy the dangling condition (see Fig. 1b). Then the following defines a graph D, the deletion of L and G according to d.*

1. *$V_D = V_G - g_V(V_L - b'_V(V_K))$ and $E_D = E_G - g_E(E_L - b'_E(E_K))$ induce the inclusion $D \to G$, and*
2. *there is an injective graph morphism $d \colon K \to D$, defined by $d(x) = g(b'(x))$ for all items x in K.*

The following definition introduces the concept of *pushouts* in the category of graphs.

Definition 8 (Pushout)**.** Given graph morphisms $b \colon A \to B$ and $c \colon A \to C$, a graph D together with graph morphisms $f \colon B \to D$ and $g \colon C \to D$ is a *pushout* of $A \to B$ and $A \to C$ if the following holds (see Fig. 2):

1. Commutativity: $f \circ b = g \circ c$
2. Universal property: For all graph morphisms $p \colon B \to H$ and $t \colon C \to H$ such that $p \circ b = t \circ c$, there is a unique morphism $u \colon D \to H$ such that $u \circ f = p$ and $u \circ g = t$. □

The formalisation of pushouts and the proof that pushouts are unique up to isomorphism is presented in Subsection 3.2.

Theorem 1 (Uniqueness of pushouts [1])**.** *Let $A \to B$ and $A \to C$ together with D induce a pushout as depicted in Fig. 2. A graph H together with morphisms $B \to H$ and $C \to H$ is a pushout of b and c if and only if there is an isomorphism $u \colon D \to H$ such that $u \circ f = p$ and $u \circ g = t$.*

Theorem 2 (Gluings are pushouts [8])**.** *Let $b \colon K \to R$ and $d \colon K \to D$ be injective graph morphisms, and H be the gluing of D and R according to d, as defined in Lemma 1. Then, the square in Fig. 1a is a pushout diagram where $D \to H$ is an inclusion and h is defined by $h(x) =$ **if** $x \in R - b(K)$ **then** x **else** $c(d(x))$. We call H the pushout object.*

The deletion construction of Lemma 2 and the following theorem are formalised and proved in Subsection 3.4.

Theorem 3 (Deletions are pushouts [8])**.** *Let $K \to L$ and $g\colon L \to G$ be injective graph morphisms and let g satisfy the dangling condition and the subgraph D of G as defined in Lemma 2. Then, the square in Fig. 1b is a pushout diagram where g is an inclusion and $d(x) = g(b'(x))$ for all items x in K. We call D the pushout complement.*

The following definition of rule application is formalised in Subsection 3.5.

Definition 9 (Direct derivation)**.** Let $r = (L \leftarrow K \rightarrow R)$ be a rule and $g\colon L \to G$ be an injective graph morphism satisfying the dangling condition. Then G directly derives (see Fig. 3) M by r and g, denoted by $G \Rightarrow_{r,g} M$, if $H \cong M$, where H is constructed from G by:

1. (Deletion) D is the subgraph $G - g(L - b'(K))$.
2. $d\colon K \to D$ is the restriction of g to K and D.
3. (Gluing) H is the gluing $H = D + (R - b(K))$. □

The following corollary follows directly by Theorem 2 and Theorem 3.

Corollary 1 (Direct derivation are double-pushouts)**.** *Given a direct derivation $G \Rightarrow_{r,g} M$, squares (1) and (2) in Figure 3 are pushouts.*

The next section provides a general introduction to the Isabelle proof assistant and highlights selected parts of our formalisation.

3 DPO Formalisation in Isabelle/HOL

Isabelle is a generic, interactive theorem prover based on the so-called *LCF* approach. It is based on a small (meta-logical) proof kernel, which is responsible for checking all proofs. This concept provides high confidence in the prover's soundness. Isabelle/HOL refers to the higher-order logic instantiation which is considered to be the most established calculus within the Isabelle distribution [18].

In Isabelle, type variables are denoted by a leading apostrophe. A term f of type $'a$ is denoted by $f::'a$. The function type from $'a$ to $'b$ is written $f::'a \Rightarrow 'b$. The inference rule notation $[\![A_1; A_2]\!] \Longrightarrow C$ with premises A_1 and A_2 and conclusion C is a shorthand (with the ; (semicolon) as a logical *and*) for the implication $A_1 \Longrightarrow A_2 \Longrightarrow C$. Its natural representation is given by:

$$\frac{A_1 \qquad A_2}{C}$$

Isabelle meta-logical universal quantifier $\bigwedge$ corresponds to HOL's $\forall$ and the meta implication $\Longrightarrow$ to $\longrightarrow$. The meta logic is used to expressed inference rules and cannot appear in HOL formulae.

Our formalisation is based on Isabelle's **locale** mechanism, a technique for writing parametric specifications. Furthermore, we use *intelligible semi-automated reasoning* (Isar) which is Isabelle's language of writing structured proofs [27]. In contrast to *apply-scripts*, which execute deduction rules in a linear manner, Isar follows a structured approach resulting in increased readability and maintainability [16].

A general introduction to Isabelle/HOL can be found in [16]. The main components of our formalisation and their interdependencies are depicted in Fig.4. The simple arrow ($\rightarrow$) can be read as "depends on", i.e., the definition of morphisms depends on the definition of graphs allowing the inheritance of properties.

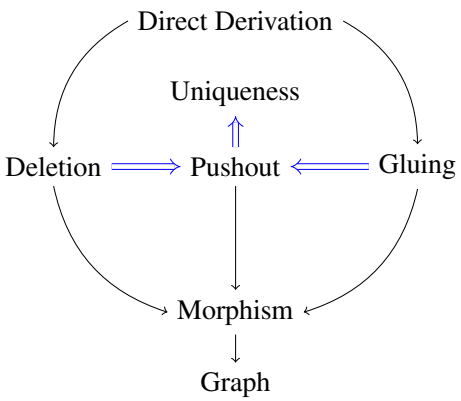


Figure 4: Overview of component dependencies ($\rightarrow$) and major theorems ($\Rightarrow$)

The blue arrow $\Longrightarrow$ highlights main theorems proven in this study, viz. that the gluing and deletion constructions correspond to pushouts and that pushout objects are unique up to isomorphism.

The upcoming subsection introduces the basic building blocks of our formalisation: Graphs and graph morphisms.

3.1 Graphs and Morphisms

Our definition of graphs (Def. 2) is different from Strecker's [23] where a graph is a set of nodes together with a binary relation of nodes. A consequence of Strecker's definition is the absence of parallel edges and edge labels. We follow Noschinski's graph library [17] approach in the sense, that we use a **record** data structure to represent a graph and enforce the well-formedness by using the **locale** mechanism. We extend Noschinski's data structure to carry node and edge labelling functions.

The usage of type variables for node and edge identifiers ($'v$ and $'e$), and labels ($'l$ and $'m$) allows us to reason about an arbitrary representation. Common examples include natural numbers (*nat*) and strings (*string*).

We define graphs using the **record** keyword as follows:

record $('v,'e,'l,'m)$ *pre-graph* =
  nodes :: $'v$ *set*
  edges :: $'e$ *set*
  source :: $'e \Rightarrow 'v$
  target :: $'e \Rightarrow 'v$
  node-label :: $'v \Rightarrow 'l$
  edge-label :: $'e \Rightarrow 'm$

With the **abbreviation** command, term abbreviations are introduced. The (built-in) axiomatized term *undefined*::$'a$ is used to refer to a fixed but arbitrary term of type $'a$.

Following this, we can define an abbreviation G, representing the empty graph (*pre-graph*) structure as follows:

abbreviation G **where**
$\langle G \equiv (\!|nodes = \{\}, edges = \{\}, source{=}undefined, target{=}undefined$
  $,node\text{-}label{=}undefined, edge\text{-}label{=}undefined|\!)\rangle$

We introduce a notation for working with the *pre-graph* structure, closely following Definition 2, using Isabelle's **notation** keyword. Throughout the formalization, we use V_- and E_- to refer to the set of nodes and edges, s_- and t_- to refer to source and target functions, and l_- and m_- refer to the node-label an edge-label functions. This allows us, for example, to write V_G instead of *nodes G* to refer to the set of nodes of a graph G.

As the *pre-graph* record does not introduce any constraints, not only well-formed graphs can be represented but also ill-formed graphs such as a graph with edges but no nodes. The well-formedness is enforced via the *graph* locale. Here, the *fixes* keyword is used to declare parameters while the *assumes* keyword is used to state premises which hold within the locale context. The *graph* locale is defined as follows:

locale *graph* =
  fixes $G :: ('v,'e,'l,'m)$ *pre-graph*
  assumes
   finite-nodes: *finite* V_G **and**
   finite-edges: *finite* E_G **and**
   source-integrity: $e \in E_G \Longrightarrow s_G\ e \in V_G$ **and**
   target-integrity: $e \in E_G \Longrightarrow t_G\ e \in V_G$

In this formalisation, the premises are:

- The set of nodes (*finite-nodes*) and edges (*finite-edges*) are finite,
- and the source (*source-integrity*) and target (*target-integrity*) functions map each edge to a node within the graph.

We do not have to state explicit premises for both, node and edge, labelling functions as they are defined for the entire universe of the corresponding type ($'v$ and $'e$).

We can prove, the *pre-graph* structure G is indeed a *graph* according to our **locale** definition by using the **interpretation** command as follows:

interpretation *graph G*
 by *unfold-locales simp-all*

The *unfold-locales* tactic applies all introduction rules generated by the **locale** command to the current proof goal. The introduction rule for the *graph* locale is given by:

$$\frac{\textit{finite } V_G \qquad \textit{finite } E_G \qquad \bigwedge e.\ \frac{e \in E_G}{s_G\ e \in V_G} \qquad \bigwedge e.\ \frac{e \in E_G}{t_G\ e \in V_G}}{\textit{graph } G}$$

To prove our graph structure $(\!|\textit{nodes} = \emptyset, \textit{edges} = \emptyset, \textit{source} = \textit{undefined}, \textit{target} = \textit{undefined}, \textit{node-label} = \textit{undefined}, \textit{edge-label} = \textit{undefined}|\!) :: ('a, 'b, 'c, 'd)$ *pre-graph* fulfills the *graph* premises, we have to prove, the set of nodes (edges) is finite and the source (target) integrity. Isabelle's simplifier is able to discharge these goals automatically.

Our definition of graph morphisms (cf. Def. 3) follows a similar pattern. We define the graph morphism data structure $(('v_1, 'v_2, 'e_1, 'e_2)$ *pre-graph*$)$ with a dedicated function for the nodes and edges:

record $('v_1,'v_2,'e_1,'e_2)$ *pre-morph* =
  node-map :: $'v_1 \Rightarrow 'v_2$
  edge-map :: $'e_1 \Rightarrow 'e_2$

Note that, a graph morphism maps the graph structure $('v_1, 'e_1, 'c, 'd)$ *pre-graph* to $('v_2, 'e_2, 'c, 'd)$ *pre-graph*, i.e., the node and edge types change. Again, a common notation $\_V$ and $\_E$ for a morphism record is introduced using the **notation** keyword.

The locale *morphism* inherits properties from the *graph* locale via the *import* mechanism. With this, a morphism carries its domain (G: *graph G*) and its codomain (H: *graph H*) and all properties (i.e., all *graph*, specialized for the particular instance), are inherited. The *pre-morph* record type is used to introduce a locale parameter f, which contains the corresponding node and edge mappings.

The morphism properties are enforced by the following axioms:

- *Range restriction*, $\dfrac{e \in E_G}{f_E\ e \in E_H}$ and $\dfrac{v \in V_G}{f_V\ v \in V_H}$
- *Source and target preservation*, $\dfrac{e \in E_G}{f_V\ (s_G\ e) = s_H\ (f_E\ e)}$ and $\dfrac{e \in E_G}{f_V\ (t_G\ e) = t_H\ (f_E\ e)}$
- *Label preservation*, $\dfrac{v \in V_G}{l_G\ v = l_H\ (f_V\ v)}$ and $\dfrac{e \in E_G}{m_G\ e = m_H\ (f_E\ e)}$

The *morphism* locale definition is given by:

locale *morphism* =
  G: *graph G* +
  H: *graph H* **for**
    $G :: ('v_1, 'e_1, 'l, 'm)$ *pre-graph* **and**
    $H :: ('v_2, 'e_2, 'l, 'm)$ *pre-graph* +
  fixes
    $f :: ('v_1, 'v_2, 'e_1, 'e_2)$ *pre-morph*
  assumes
    morph-edge-range: $e \in E_G \Longrightarrow f_E\ e \in E_H$ **and**
    morph-node-range: $v \in V_G \Longrightarrow f_V\ v \in V_H$ **and**
    source-preserve : $e \in E_G \Longrightarrow f_V\ (s_G\ e) = s_H\ (f_E\ e)$ **and**
    target-preserve : $e \in E_G \Longrightarrow f_V\ (t_G\ e) = t_H\ (f_E\ e)$ **and**
    label-preserve : $v \in V_G \Longrightarrow l_G\ v = l_H\ (f_V\ v)$ **and**
    mark-preserve : $e \in E_G \Longrightarrow m_G\ e = m_H\ (f_E\ e)$

With this, we define the composition of graph morphisms (cf. Def. 5) including the infix notation $\circ_\rightarrow$ as the pairwise compositions:

definition *morph-comp*
  $:: ('v_2, 'v_3, 'e_2, 'e_3)$ *pre-morph* $\Rightarrow$ $('v_1, 'v_2, 'e_1, 'e_2)$ *pre-morph* $\Rightarrow ('v_1, 'v_3, 'e_1, 'e_3)$ *pre-morph* (**infixl** $\circ_\rightarrow$ *55*) **where**
  $g \circ_\rightarrow f = (\!|$*node-map* $= g_V \circ f_V$, *edge-map* $= g_E \circ f_E|\!)$

The proposition, from *morphism G H f* and *morphism H K g* we can conclude *morphism G K* $(g \circ_\rightarrow f)$ is expressed using the *Isar* language as follows:

lemma
  assumes f: ‹*morphism G H f*› **and** g: ‹*morphism H K g*›
  shows ‹*morphism G K* $(g \circ_\rightarrow f)$›

Each premise, indicated by the *assumes* keyword, is (optionally) associated with a name f and g, respectively. The conclusion is indicated by the *shows* statement. We enter the proof by the **proof** command

followed by an optional proof method. In this particular case, we use the *intro-locales* method, which applies the introduction rules of locales.

proof *intro-locales*

Isabelle generates the following subgoals to discharge the lemma:

1. graph G
2. graph K
3. morphism-axioms G K $(g \circ_{\to} f)$

The first two subgoals follow directly from the locale definition of *morphisms*. The proof of *graph G* is give by supplying the corresponding fact:

show ‹*graph G*› **by** (*fact morphism.axioms*[*OF f*])

The *morphism.axioms* definition is generated by the locale approach covering the stated locale assumptions. The *OF* command is used to apply one theorem to another. The subgoal *graph K* follows analogously. To prove the morphism axioms (*morphism-axioms*), the *morphism-axioms.intro* introduction rule (generated by Isabelle) is used. Its definition is as follows:

$$\frac{\bigwedge e.\ \dfrac{e \in E_G}{g \circ_{\to} f^E\ e \in E_K} \qquad \bigwedge v.\ \dfrac{v \in V_G}{g \circ_{\to} f^V\ v \in V_K} \qquad \bigwedge e.\ \dfrac{e \in E_G}{g \circ_{\to} f^V\ (s_G\ e) = s_K\ (g \circ_{\to} f^E\ e)} \qquad \bigwedge e.\ \dfrac{e \in E_G}{g \circ_{\to} f^V\ (t_G\ e) = t_K\ (g \circ_{\to} f^E\ e)} \qquad \bigwedge v.\ \dfrac{v \in V_G}{l_G\ v = l_K\ (g \circ_{\to} f^V\ v)} \qquad \bigwedge e.\ \dfrac{e \in E_G}{m_G\ e = m_K\ (g \circ_{\to} f^E\ e)}}{\textit{morphism-axioms}\ G\ K\ (g \circ_{\to} f)}$$

Both, the constant (*morphism-axioms*) rule and the introduction rule, are generated by the **locale** mechanism.

show ‹*morphism-axioms G K* $(g \circ_{\to} f)$›
proof

The **proof** command, without an explicit proof method will use the *standard* method. This method uses a heuristic to apply certain proof rules. In this particular case, the introduction rule is used which results the following subgoals:

1. $\bigwedge e.\ e \in E_G \Longrightarrow g \circ_{\to} f^E\ e \in E_K$
2. $\bigwedge v.\ v \in V_G \Longrightarrow g \circ_{\to} f^V\ v \in V_K$
3. $\bigwedge e.\ e \in E_G \Longrightarrow g \circ_{\to} f^V\ (s_G\ e) = s_K\ (g \circ_{\to} f^E\ e)$
4. $\bigwedge e.\ e \in E_G \Longrightarrow g \circ_{\to} f^V\ (t_G\ e) = t_K\ (g \circ_{\to} f^E\ e)$
5. $\bigwedge v.\ v \in V_G \Longrightarrow l_G\ v = l_K\ (g \circ_{\to} f^V\ v)$
6. $\bigwedge e.\ e \in E_G \Longrightarrow m_G\ e = m_K\ (g \circ_{\to} f^E\ e)$

Exemplary, we show that the composition $g \circ_{\to} f$ maps an edge from *G* to an edge from *K*. This subgoal arises from the *morph-edge-range* axiom.

show ‹$g \circ_{\to} f^E\ e \in E_K$› **if** ‹$e \in E_G$› **for** *e*
by (*simp add*: *morph-comp-def morphism.morph-edge-range*[*OF g*] *morphism.morph-edge-range*[*OF f*] *that*)

To prove this goal, we unfold the definition of $(\circ_{\to})$ by telling the simplifier to consider the morphism composition definition (*morph-comp-def*) fact. With the fact that both, the *morph-edge-range* axiom hold for *g* and *f*, and built-in facts on function composition, the simplifier is able to discharge the goal.

Proving that the composition preserves the sources follows similarly. We unfold the composition definition and supply the *morph-edge-range* and *source-preserve*, specialized for each morphism f and g to the simplifier:

show ‹$s_{g \circ_\to f^V}\ (s_G\ e) = s_K\ (_{g \circ_\to f^E}\ e)$› **if** ‹$e \in E_G$› **for** e
by (*simp add*: *morph-comp-def*
morphism.morph-edge-range[*OF f*]
morphism.morph-edge-range[*OF g*]
morphism.source-preserve[*OF f*]
morphism.source-preserve[*OF g*] *that*)

Isabelle's simplifier is able to discharge the proof obligation with the supplied facts. The other subgoals follow analogously.

Based on the *morphism* locale, we formalise injective graph morphisms (cf. Def. 4) in a separate locale as follows:

locale *injective-morphism* = *morphism* +
assumes
inj-nodes: *inj-on* $_{f^V}\ V_G$ **and**
inj-edges: *inj-on* $_{f^E}\ E_G$

The locale axioms (*inj-nodes* and *inj-edges*) are used to restrict the node and edge mappings to be injective over the corresponding domain using Isabelle's built-in *inj-on* predicate. Furthermore, we define surjective and bijective morphisms in a similar way.

3.2 Pushouts

This subsection formalises pushouts in Isabelle and proves their uniqueness up to isomorphism (cf. Theorem 1). The pushout characterisation comprises four commuting morphisms ($A \to B$, $A \to C$, $B \to D$, and $C \to D$) which satisfy the universal property (cf. Def. 8). The commuting property is expressed using the node *node-commutativity* and edge *edge-commutativity* proposition and the composition of morphisms. Our formalisation is given by:

locale *pushout-diagram* =
b: *morphism A B b* +
c: *morphism A C c* +
f: *morphism B D f* +
g: *morphism C D g* **for** *A B C* **and** *D* :: ‹('*g*, '*h*, '*k*, '*l*) *pre-graph*› **and** *b c f g* +
assumes
node-commutativity: ‹$v \in V_A \Longrightarrow f \circ_\to b^V\ v = g \circ_\to c^V\ v$› **and**
edge-commutativity: ‹$e \in E_A \Longrightarrow f \circ_\to b^E\ e = g \circ_\to c^E\ e$› **and**
universal-property: ‹⟦
graph (*D'* :: ('*g*,'*h*,'*k*,'*l*) *pre-graph*);
morphism B D' x;
morphism C D' y;
$\forall v \in V_A \cdot\ x \circ_\to b^V\ v = y \circ_\to c^V\ v$;
$\forall e \in E_A \cdot\ x \circ_\to b^E\ e = y \circ_\to c^E\ e$⟧
$\Longrightarrow$ *Ex1M* (λu. *morphism D D' u* $\wedge$
$(\forall v \in V_B \cdot\ u \circ_\to f^V\ v = x^V\ v) \wedge$
$(\forall e \in E_B \cdot\ u \circ_\to f^E\ e = x^E\ e) \wedge$
$(\forall v \in V_C \cdot\ u \circ_\to g^V\ v = y^V\ v) \wedge$

$(\forall e \in E_C \cdot u \circ_{\to} g_E\ e = y_E\ e))$
$D\rangle$

In Isabelle, unbound variables are implicitly bound using the (meta) universal quantifier. For a given P, the proposition $P\ x$ is interpreted as $\bigwedge x.\ P\ x$. Additionally, we restrict the universal quantified *pre-graph* record to match the pushout object (graph D) type. This prevents a warning generated by Isabelle's locale mechanism of newly introduced type-parameter. We discuss this implication in Section 4.

Compared to our initial version of this paper [22], the usage of total functions increased the complexity of the *pushout-diagram* definition. The *universal-property*, stating the existence of a unique morphisms (cf. Definition 8), cannot use the built-in unique existence operator (*Ex1*). The proof of this operator would result in a goal to prove equality of the morphism u from D to D'. As function equality (i.e. $u = u'$) requires equality across the universe of values of the domain (using the built-in simplification rule *fun-eq-iff*, which is defined as $(u = u') = (\forall x.\ u\ x = u'\ x)$). In our formalisation, equality over the entire domain is not true. We discuss this implication in Section 4.

Our solution quantifies over the set of values (i.e., the set of nodes and edges of the source graph). Therefore, we introduce the abbreviation

$Ex1M \equiv$
$\lambda P\ E.\ \exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow (\forall e{\in}E_E \cdot\ y_E\ e = x_E\ e) \wedge (\forall v{\in}V_E \cdot\ y_V\ v = x_V\ v))$

which is a lambda term capturing the predicate P and the source graph E to quantify over the corresponding sets.

To prove the uniqueness of the pushout object, we assume the *pushout-diagram* of $A \to B, A \to C, B \to D$, and $C \to D$, then the graph D' together with two morphisms $f' : B \to D'$ and $g' : C \to D'$ is a pushout if and only if there exists a bijection u between D and D' such that the triangles commute ($\forall x \in B.u \circ fx = f'x$ and $\forall x \in D.u \circ gx = g'x$).

We formalise this theorem in Isabelle as follows:

theorem *uniqueness-po*:
fixes D' :: ‹('g, 'h, 'k, 'l) *pre-graph*›
assumes
D': ‹*graph* D'› **and**
f': ‹*morphism* $B\ D'\ f'$› **and**
g': ‹*morphism* $C\ D'\ g'$›
shows ‹*pushout-diagram* $A\ B\ C\ D'\ b\ c\ f'\ g'$
$\longleftrightarrow$ ($\exists u$. *bijective-morphism* $D\ D'\ u$
$\wedge$ $(\forall v \in V_B \cdot u \circ_{\to} f'_V\ v = f'_V\ v) \wedge (\forall e \in E_B \cdot u \circ_{\to} f'_E\ e = f'_E\ e)$
$\wedge$ $(\forall v \in V_C \cdot u \circ_{\to} g'_V\ v = g'_V\ v) \wedge (\forall e \in E_C \cdot u \circ_{\to} g'_E\ e = g'_E\ e))$›
proof

The (implicitly) applied proof method *standard* applies the built-in introduction rule *iffI*, which is used for *if and only if* proofs, resulting in the following two subgoals:

1. *pushout-diagram* $A\ B\ C\ D'\ b\ c\ f'\ g' \Longrightarrow$
$\exists u$. *bijective-morphism* $D\ D'\ u \wedge$
$(\forall v{\in}V_B \cdot u \circ_{\to} f'_V\ v = f'_V\ v) \wedge$
$(\forall e{\in}E_B \cdot u \circ_{\to} f'_E\ e = f'_E\ e) \wedge$
$(\forall v{\in}V_C \cdot u \circ_{\to} g'_V\ v = g'_V\ v) \wedge (\forall e{\in}E_C \cdot u \circ_{\to} g'_E\ e = g'_E\ e)$
2. $\exists u$. *bijective-morphism* $D\ D'\ u \wedge$

$(\forall v{\in}V_B{\cdot}\ u \circ_\to f_V\ v = {f'}_V\ v) \wedge$
$(\forall e{\in}E_B{\cdot}\ u \circ_\to f_E\ e = {f'}_E\ e) \wedge$
$(\forall v{\in}V_C{\cdot}\ u \circ_\to g_V\ v = {g'}_V\ v) \wedge (\forall e{\in}E_C{\cdot}\ u \circ_\to g_E\ e = {g'}_E\ e) \Longrightarrow$
pushout-diagram A B C D' b c f' g'

The proofs of both subgoals are omitted, but can be found on the GitHub repository for this formalisation. In the upcoming section, we will describe our formalisation of the *gluing* construction.

3.3 Gluings are Pushouts

The locale *gluing* is used as an environment with required preconditions, i.e., two injective morphisms as described in Lemma 1. Our definition is as follows:

locale *gluing* =
d: *injective-morphism K D d* +
r: *injective-morphism K R b*
for *K D R d b*

Within the locales context, we first define the gluing construction (graph D together with injective morphisms h and c) as depicted in Fig. 1a. Subsequently, we prove pushout correspondence (cf. Theorem 2) by interpretation of the *pushout-diagram* locale. Note that, while the **interpretation** command is used for temporal instantiations (limited to the current context block), the **sublocale** command is used to create persistent links between locales (see [26]). We use this technique to prove Corollary 1.

The gluing graph D can be constructed using the disjoint union of the node (edge) set (cf. Lemma 1). In our formalisation, we use the built-in sum type $'a + 'b$ type, it comes with the two injective functions $Inl::'a \Rightarrow 'a + 'b$ and $Inr::'b \Rightarrow 'a + 'b$ which correspond to the i_A and i_B, see Section 2. The image of a set A under a function f is denoted by f ‘ A (*backtick* operator).

The node set is constructed by using the image of the nodes of D under the injection *Inl* united with the image of the nodes R without b ‘ K under the injection *Inr*:

abbreviation *V* **where** ‹$V \equiv Inl$ ‘ $V_D \cup Inr$ ‘ $(V_R - {b}_V$ ‘ $V_K)$›

The edge set follows analogously. We use the **fun** command to state the source (target) function, as well as the labelling functions. It will try proving certain properties (e.g., termination) of the function under investigation automatically. In case the automation fails, the user has to prove these properties by hand.

In our cases, Isabelle is able to discharge all generated proof obligations automatically.

For the definition of the source (target) function, the general idea is a case analysis on the edge origin by pattern matching on *Inl e* and *Inr e* constructors. The source function is defined by:

fun *s* **where**
$s\ (Inl\ e) = Inl\ (s_D\ e)$
$|s\ (Inr\ e) = (if\ e \in (E_R - {b}_E$ ‘ $E_K) \wedge (s_R\ e \in {b}_V$ ‘ $V_K)$
then Inl $({}_{d}V\ ((inv\text{-}into\ V_K\ {b}_V)\ (s_R\ e)))$ *else Inr* $(s_R\ e))$

In the *Inl e* case, by definition of the edge set, the edge e belongs to the graph D. As a result, we use the source function of D. Compared to Section 2, where we assume i_A (i_B) are inclusions (to keep the section readable), we now have to be more explicit. In the *Inr e* case, the inverse of b is given by the built-in *inv-into* function.

The target function is analogous. Both labelling functions (for nodes and edges) are defined by case analysis of the origin. Exemplary, the node labelling function l is given by:

fun l **where**
$l\ (Inl\ v) = l_D\ v$
$|\ l\ (Inr\ v) = l_R\ v$

The edge labelling function is defined analogously. We follow by proving that these elements indeed form a graph (according to our locale *graph*) by interpretation. The proof is mechanic and follows by case splitting on the sum type.

Further, we define the morphisms $h\colon R \to H$ and $c\colon D \to H$. The morphism h is defined by a case distinction of the node (edge) parameter. If the node (edge) v is in the set $V_R - {}_{bV}$ ‘ V_K we know it is a newly created node (edge) and therefore has to be lifted using the *Inr* injection. Otherwise, the node (edge) is already in the graph D. Due to the injective b (compared to inclusion), we use the inverse of b followed by d. Finally, the node (edge) is lifted into the sum type by using the *Inl* injection.

Our definition is as follows:

abbreviation h **where**
$\langle h \equiv (\!|node\text{-}map = \lambda v.\ if\ v \in V_R - {}_{bV}$ ‘ $V_K\ then\ Inr\ v\ else\ Inl\ ({}_{dV}\ ((inv\text{-}into\ V_K\ {}_{bV})\ v)),$
$edge\text{-}map = \lambda e.\ if\ e \in E_R - {}_{bE}$ ‘ $E_K\ then\ Inr\ e\ else\ Inl\ ({}_{dE}\ ((inv\text{-}into\ E_K\ {}_{bE})\ e))|\!)\rangle$

The morphism c is defined by the node (edge) injection *Inl*:

abbreviation c **where** $\langle c \equiv (\!|node\text{-}map = Inl, edge\text{-}map = Inl|\!)\rangle$

Each time, we prove that our construction fulfills the *injective-morphism* axioms by interpretation. The proofs can be found on GitHub (interpretation *inj-h* and *inj-c*). Finally, we are able to prove the pushout correspondence by instantiating (using the **sublocale** command) of the *pushout-diagram* locale with the corresponding parameters:

sublocale *po*: *pushout-diagram K R D H b d h c*

Here, *po* is used to refer to *pushout-diagram* instance. The proof is around 170 lines of text.

In the upcoming subsection, we describe our formalisation of the *deletion* construction.

3.4 Deletions are Pushouts

The *deletion* locale is also used as an environment with the required preconditions as stated in Lemma 2. The dangling condition (cf. Def. 7) is expressed using separate rules for the source (*dang-src*) and target (*dang-trg*) mapping as follows:

$$\frac{e \in E}{s_G\ e \notin {}_{gV}\ \text{‘}\ (V_L - {}_{b'V}\ \text{‘}\ V_K)} \qquad \frac{e \in E}{t_G\ e \notin {}_{gV}\ \text{‘}\ (V_L - {}_{b'V}\ \text{‘}\ V_K)}$$

These locale assumptions are introduced in the *assumes* section as follows:

locale *deletion* =
g: *injective-morphism L G g* +
l: *injective-morphism K L b′*
for *K G L g b′* +
assumes
dang-src: $\langle e \in E_G - {}_{gE}$ ‘ $(E_L - {}_{b'E}$ ‘ $E_K) \Longrightarrow s_G\ e \notin {}_{gV}$ ‘ $(V_L - {}_{b'V}$ ‘ $V_K)\rangle$ **and**
dang-trg: $\langle e \in E_G - {}_{gE}$ ‘ $(E_L - {}_{b'E}$ ‘ $E_K) \Longrightarrow t_G\ e \notin {}_{gV}$ ‘ $(V_L - {}_{b'V}$ ‘ $V_K)\rangle$

The construction of the graph D removes all nodes (edges) from G which belong to a subgraph L (without the image of b under K) under the morphism g. The construction of the node set is given by:

abbreviation V **where** $\langle V \equiv V_G - {}_{gV} \text{`} (V_L - {}_{b'V} \text{`} V_K)\rangle$

The edge set follows analogously. The graph D (*pre-graph* record) is constructed by updating graph G with a restricted set of nodes and edges. We use the record update syntax $G(\!|nodes{:=}V, edges{:=}E|\!)$, which replaces the set of nodes (edges) by V (E). The other graph elements (i.e., s, t, l, and m) do not have to be changed, as we quantify over the corresponding node (edge) set and therefore limit our reasoning to the valid subset of nodes (edges).

We define the injective morphisms $d\colon K \to D$ by the composition of g after b':

abbreviation d **where**
$\langle d \equiv g \circ_{\to} b'\rangle$

The *injective-morphism* interpretation (*inj-d*) can be found on GitHub. The proof relies on the specialised proposition of well-formed composition of graph morphisms (*wf-morph-comp*).

The morphism $c'\colon D \to G$ is an inclusion, which we define using the *idM*::('*i*, '*i*, '*j*, '*j*) *pre-morph* as:

abbreviation c' **where** $\langle c' \equiv idM\rangle$

The *idM* record (defined in the *DPO−Formal.Morphism* theory) uses the built-in identity function (*id*::'*i* $\Rightarrow$ '*i*) for both, the node and edge mappings (i.e., $(\!|$*node-map* $= id$, *edge-map* $= id|\!)$).

Here, the *injective-morphism* interpretation can be solved by the simplifier using the *simp-all* proof method.

Ultimately, we prove the pushout correspondence by interpretation of the *pushout-diagram* locale with the corresponding parameters:

sublocale *po*: *pushout-diagram K L D G b' d g c'*

The proof is around 350 lines of text and follows mainly by case analysis of the edge (node) origin. A concise proof, relying on the gluing construction of D and L, failed due to Isabelle's typechecker not accepting our definition. Our *gluing* construction results in a pushout object with the type signature ('*a* + '*e*, '*b* + '*f*, '*c*, '*d*) *pre-graph*. As a result, we cannot use the universal property for graph G (with signature ('*e*, '*f*, '*c*, '*d*) *pre-graph*) as these types mismatch. We will comment in Section 4 on this.

In the following subsection we introduce rules and the notation of direct derivations.

3.5 Rules and Derivations

Our formalisation of rules (cf. Def. 6) relies on the inclusion $K \to L$ and $K \to R$ which we represent in the locale *rule*:

locale *rule* =
  k: *inclusion-morphism K L* +
  r: *inclusion-morphism K R*
  for *L K R*
begin

As we have explicitly included the domain and codomain in our definition of morphisms, we inherit all properties and Isabelle's locale mechanism allows this dense definition.

Using the **notation** we introduce the common syntactical representation $L \leftarrow K \rightarrow R$ for L, K, and R of type $('v, 'e, 'l, 'm)$ *pre-graph* on the *rule* predicate.

A direct derivation (cf. Def. 9) is defined in terms of existing constructs; the *direct-derivation* locale imports from the *rule* and passes the pushout complement from the *deletion* locale ($d.H$) into the *gluing* locale to construct pushout object $g.h$. We model this in Isabelle as follows:

locale *direct-derivation* =
  r: *rule L K R* +
  d: *deletion K G L g idM* +
  g: *gluing K d.D R g idM* **for** *G L K R g*

Within the *direct-derivation* context, we prove that given a direct derivation $G \Rightarrow_{r,g} M$, squares (1) and (2) in Figure 3 are pushouts (cf. Corollary 1). We state this corollary as follows:

corollary
  ‹*pushout-diagram K L d.D G idM d.d g d.c'*› **and** ‹*pushout-diagram K R d.D g.H idM g g.h g.c*›
 using
  d.po.pushout-diagram-axioms
  g.po.pushout-diagram-axioms
 by *simp-all*

Isabelle's simplifier is able to discharge the proof obligations after supplying the corresponding facts. In the upcoming section we state our conclusion and areas of future work.

4 Conclusion and Future Work

Formal verification increases the trustworthiness and reliability of software. In this paper, we present a revised version of [22] on our formalisation of the double-pushout approach with injective matching over (node and edge) labelled directed graphs, in the proof assistant Isabelle/HOL. The formalisation uses the extensible locale mechanism, which allows us to combine theories and to structure our work. Compared to earlier work in [22], we rely on total functions and follow Noschinski's approach of representing graphs and morphisms. With these changes, we reduced the required lines of text by 50 percent while enhancing overall readability.

We first formalise graphs and morphisms and prove several properties, such as the well-definedness of morphism composition. Direct derivations are introduced in terms of deletion and gluing. We prove their correspondence to pushouts. In addition, we prove that pushouts are unique up to isomorphism. Although, Isabelle is not able to discharge most of the generated proof obligations automatically, the available proof methods support the discharging process.

Our formalisation of pushouts results in pushout objects of a to specific type, which prevents us from constructing an elegant proof within the deletion context (as we highlight at the end of Subsection 3.4). In the meantime, we have overcome this limitation by imposing the constraint that nodes and edges must be natural numbers (only within the universal property). This is realised by a type synonym such as **type-synonym** $('l,'m)$ *ngraph* $=$ $(nat,nat,'l,'m)$ *pre-graph*. By restricting our locale type variables to have an instance of the *countable* class, we could define functions to convert between the natural and generic representation. If $'a$ is an instance of the *countable* class (denoted by $'a :: countable$), there exists an injective function, mapping each element of type $'a$ to an element of typ *nat*.

Future developments will also include the proof of classical DPO results such as the Church-Rosser theorem. We also plan the extension towards attributed DPO graph transformation. Our long-term goal is the development of a practical Isabelle-based proof assistant for the verification of individual programs in the graph programming language GP 2 [5].

References

[1] Jiří Adámek, Horst Herrlich & George Strecker (1990): *Abstract and Concrete Categories*. Wiley.

[2] Luciano Baresi, Vahid Rafe, Adel T. Rahmani & Paola Spoletini (2008): *An Efficient Solution for Model Checking Graph Transformation Systems*. Electronic Notes in Theoretical Computer Science 213(1), pp. 3–21, doi:10.1016/j.entcs.2008.04.071.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, Springer, doi:10.1007/978-3-662-07964-5.

[4] Jon Haël Brenas, Rachid Echahed & Martin Strecker (2018): *Verifying graph transformations with guarded logics*. In: *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, IEEE, pp. 124–131, doi:10.1109/TASE.2018.00024.

[5] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs*. Science of Computer Programming 214, doi:10.1016/j.scico.2021.102727.

[6] Simone André da Costa Cavalheiro, Luciana Foss & Leila Ribeiro (2017): *Theorem proving graph grammars with attributes and negative application conditions*. Theoretical computer science 686, pp. 25–77, doi:10.1016/j.tcs.2017.04.010.

[7] Mark Dowson (1997): *The Ariane 5 software failure*. ACM SIGSOFT Software Engineering Notes 22(2), p. 84, doi:10.1145/251880.251992.

[8] Hartmut Ehrig (1979): *Introduction to the Algebraic Theory of Graph Grammars*. In: Proc. Graph-Grammars and Their Application to Computer Science and Biology, Lecture Notes in Computer Science 73, Springer-Verlag, pp. 1–69, doi:10.1007/BFb0025714.

[9] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, Springer, doi:10.1007/3-540-31188-2. Available at https://link.springer.com/book/10.1007%2F3-540-31188-2.

[10] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and analysis using GROOVE*. International Journal on Software Tools for Technology Transfer 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[11] Annegret Habel, Jürgen Müller & Detlef Plump (2001): *Double-Pushout Graph Transformation Revisited*. Mathematical Structures in Computer Science 11(5), pp. 637–688, doi:10.17/S0960129501003425.

[12] Annegret Habel & Karl-Heinz Pennemann (2009): *Correctness of high-level transformation systems relative to nested conditions*. Mathematical Structures in Computer Science 19(2), pp. 245–296, doi:10.1017/S0960129508007202.

[13] J. Harrison (2003): *Formal verification at Intel*. In: Proc. 18th Annual IEEE Symposium of Logic in Computer Science, IEEE Computer Society, doi:10.1109/lics.2003.1210044.

[14] Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato & Keisuke Nakano (2011): *Graph-Transformation Verification Using Monadic Second-Order Logic*. In: Proc. of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP '11), ACM Press, p. 17–28, doi:10.1145/2003476.2003482.

[15] N.G. Leveson & C.S. Turner (1993): *An investigation of the Therac-25 accidents*. Computer 26(7), pp. 18–41, doi:10.1109/mc.1993.274940.

[16] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics — With Isabelle/HOL*. Springer, doi:10.1007/978-3-319-10542-0.

[17] Lars Noschinski (2015): *A graph library for Isabelle*. *Mathematics in Computer Science* 9(1), pp. 23–39.

[18] Lawrence C. Paulson, Tobias Nipkow & Makarius Wenzel (2019): *From LCF to Isabelle/HOL*. *Formal Aspects of Computing* 31(6), pp. 675–698, doi:10.1007/s00165-019-00492-1.

[19] Christopher M. Poskitt & Detlef Plump (2014): *Verifying Monadic Second-Order Properties of Graph Programs*. In: *Proc. International Conference on Graph Transformation (ICGT 2014)*, *Lecture Notes in Computer Science* 8571, Springer, pp. 33–48, doi:10.1007/978-3-319-09108-2_3.

[20] Arend Rensink (2008): *Explicit State Model Checking for Graph Grammars*. In: *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, *Lecture Notes in Computer Science* 5065, Springer, pp. 114–132, doi:10.1007/978-3-540-68679-8_8.

[21] Arend Rensink, Ákos Schmidt & Dániel Varró (2004): *Model checking graph transformations: A comparison of two approaches*. In: *Proc. International Conference on Graph Transformation (ICGT 2004)*, Springer, pp. 226–241, doi:10.1007/978-3-540-30203-2_17.

[22] Robert Söldner & Detlef Plump (2022): *Towards Mechanised Proofs in Double-Pushout Graph Transformation*. In: *Proc. International Workshop on Graph Computation Models (GCM 2022)*.

[23] Martin Strecker (2018): *Interactive and automated proofs for graph transformations*. *Mathematical Structures in Computer Science* 28(8), pp. 1333–1362, doi:10.1017/S096012951800021X.

[24] Jan Stückrath (2016): *Verification of Well-Structured Graph Transformation Systems*. Ph.D. thesis, Universität Duisburg-Essen.

[25] Dániel Varró (2004): *Automated formal verification of visual modeling languages by model checking*. *Software & Systems Modeling* 3(2), pp. 85–113, doi:10.1007/s10270-003-0050-x.

[26] Makarius Wenzel: *The Isabelle/Isar Reference Manual*. https://isabelle.in.tum.de/doc/isar-ref.pdf.

[27] Markus Wenzel (1999): *Isar — A Generic Interpretative Approach to Readable Formal Proof Documents*. In: *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Springer, pp. 167–183.

[28] Gia S. Wulandari & Detlef Plump (2021): *Verifying Graph Programs with Monadic Second-Order Logic*. In: *Proc. International Conference on Graph Transformation (ICGT 2021)*, *Lecture Notes in Computer Science* 12741, Springer, pp. 240–261, doi:10.1007/978-3-030-78946-6_13.