Operational Equivalence of Graph Transformation Systems

Frank Raiser and Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
{Frank.Raiser|Thom.Fruehwirth}@uni-ulm.de

Abstract. Graph transformation systems (GTS) provide an important theory for numerous applications. With the growing number of GTS-based applications the comparison of operational equivalence of two GTS becomes an important area of research. This work introduces a notion of operational equivalence for graph transformation systems. The embedding of GTS in constraint handling rules (CHR) provides the basis for a decidable and sufficient criterion for operational equivalence of GTS. It is based on the operational equivalence test for CHR programs. A direct application of adapting this test to GTS allows automatic removal of redundant rules.

1 Introduction

Graph transformation systems (GTS) describe complex structures and systems in an expressive and versatile way. The principal idea of graph transformation systems is to apply graph production rules to a host graph. This involves finding a subgraph that matches the rule's graph and that is modified according to that rule.

As more applications based on graph transformations emerge it is becoming an important area of research to compare two GTS with each other. This comparison can be used to check whether two GTS solve the same problem, to verify that an optimized version of a GTS adheres to a non-optimized, but simpler, specification-GTS, to remove redundant rules, and more.

Different mechanisms for rewriting graphs have been developed [1] and in this work we make use of the so-called DPO approach [2]. It provides a sound category theoretical basis for modeling graph transformations.

Example 1. In Fig. 1 a graph transformation system consisting of two rules is shown. The graphical notation is explained in more detail later. The two rules replace edges of type a by edges of type b. An application of the operational equivalence, presented in Sect. 3.1, is the automatic removal of the second rule due to its redundancy.

In [3] an embedding of GTS in constraint handling rules (CHR) [4, 5] is given. In the context of CHR, operational equivalence is already an active area of research [6]. Therefore, we apply the results on operational equivalence in CHR

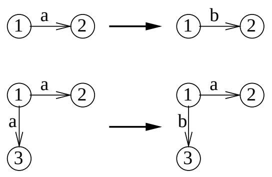

Fig. 1. Graph transformation system – the second rule is redundant

to embedded graph transformation systems. This gives us a first notion of operational equivalence for GTS and at the same time a decidable and sufficient criterion for it. In CHR research a successful application of the operational equivalence test is the removal of redundant rules [7, 8]. By building upon the GTS embedding and operational equivalence of CHR, we show that this application can be adapted to remove redundant GTS rules as well.

Note that there is an orthogonal notion for *behavioral equivalence* of graph transformation systems [9]. Behavioral equivalence investigates the behavior of models, i.e. host graphs, which are transformed into new models while preserving behavior as given by a semantics based on another graph transformation system. Similarly, in [10] bisimilarity for GTS is discussed. While bisimilarity originated from process calculi and is focused on the transitions made during computation of a result, our approach only compares the final computation results independently of how they are reached.

This paper is a preliminary work for establishing the notion of operational equivalence on graph transformation systems. We make the following contributions:

– We present definitions of joinability and operational equivalence in GTS transferred from CHR research.
– We show that given our existing embedding of GTS into CHR allows to reuse the operational equivalence test as a sufficient criterion for operational equivalence of two GTS.
– We apply the previous result to automatically remove redundant rules from a GTS.
– We observe the problem of a direct formulation of our criterion in the GTS context.

This work is divided into the following sections: Section 2 presents the necessary preliminaries for graph transformation systems, constraint handling rules, the embedding of GTS in CHR, and operational equivalence in CHR. We then introduce operational equivalence for GTS in Sect. 3, before concluding in Sect. 4.

2 Preliminaries

In this section we introduce the necessary preliminaries for this work. As our work is derived from results in different research areas, each of the following sections introduces a particular research topic. In Sect. 2.1 we introduce Graph Transformation Systems (GTS) and in Sect. 2.2 Constraint Handling Rules (CHR). Section 2.3 discusses the embedding of a GTS in CHR. Finally, Sect. 2.4 highlights existing achievements on operational equivalence for CHR programs.

2.1 Graph Transformation Systems

The following definitions for graphs and graph transformation systems have been adapted from [2].

Definition 1 (type graph, typed graph).
A graph $G = (V, E, \mathrm{src}, \mathrm{tgt})$ *consists of a finite set* V *of nodes, a finite set* E *of edges and two morphisms* $\mathrm{src}, \mathrm{tgt} : E \to V$ *specifying source and target of an edge, respectively. A* type graph TG *is a graph with unique labels for all nodes and edges.*

For multiple graphs we refer to the node set V *of a graph* G *as* V_G *and analogously for edge sets and the* $\mathrm{src}, \mathrm{tgt}$ *morphisms. We further define the degree of a node as* $\deg : V \to \mathbb{N}, v \mapsto \#\{e \in E \mid \mathrm{src}(e) = v\} + \#\{e \in E \mid \mathrm{tgt}(e) = v\}$. *When the context graph is clear the subscript is omitted.*

A typed graph G *is a tuple* $(V, E, \mathrm{src}, \mathrm{tgt}, \mathrm{type}, TG)$ *where* $(V, E, \mathrm{src}, \mathrm{tgt})$ *is a graph,* TG *a type graph, and* type *a morphism with* $\mathrm{type} = (\mathrm{type}_V, \mathrm{type}_E)$ *and* $\mathrm{type}_V : V \to V_{TG}, \mathrm{type}_E : E \to E_{TG}$. *The* type *morphism is a graph morphism, therefore, it has to satisfy the following condition:* $\forall e \in E : \mathrm{type}_V(\mathrm{src}(e)) = \mathrm{src}_{TG}(\mathrm{type}_E(e)) \wedge \mathrm{type}_V(\mathrm{tgt}(e)) = \mathrm{tgt}_{TG}(\mathrm{type}_E(e))$

For the purpose of simplicity, the above definition avoids an additional label morphism in favor of identifying variable names with labels. As there are often multiple graphs containing the same node due to inclusion morphisms we use $\deg_G(v)$ to specify the degree of a node v with respect to the graph G.

Example 2. Figure 2 shows an example for a type graph and a corresponding typed graph. The type graph at the bottom is used to define a node type and two edge types a and b. The typed graph at the top left assigns a unique node type (or edge type) to each node (or edge) via the type morphism represented by the dotted lines. The typed graph at the top right uses a shorter notation that implicitly defines the type morphism by specifying the corresponding labels.

Definition 2 (GTS, rule).
A Graph Transformation System *(GTS) is a tuple consisting of a type graph and a finite set of graph production rules. A* graph production rule *– also simply called* rule *if the context is clear – is a tuple* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *of graphs* $L, K,$ *and* R *with inclusion morphisms* $l : K \to L$ *and* $r : K \to R$.

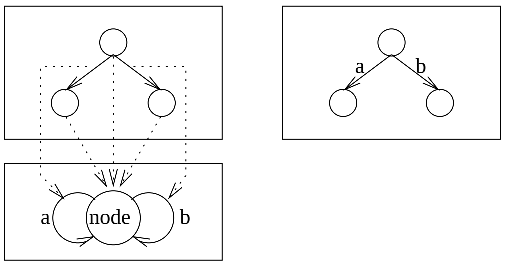

Fig. 2. Example of a type graph and typed graphs

We distinguish two kinds of typed graphs: *rule graphs* and *host graphs*. Rule graphs are the graphs L, K, R of a graph production rule p and host graphs are graphs to which the graph production rules can be applied. We, furthermore, make use of graph transformations based on the double-pushout approach (DPO) as defined in [2]. Most notably, we require a so-called match morphism $m : L \rightarrow G$ to apply a rule p to a typed host graph G. In this work we only consider injective match morphisms (see [3]). The transformation yielding the typed graph H is written as $G \overset{p,m}{\Longrightarrow} H$. H is given mathematically by constructing D as shown in Fig. 3, such that (1) and (2) are pushouts in the category of typed graphs [2]. Intuitively, the graph L on the left-hand side is matched as a subgraph of G and its occurrence in G is then replaced by the right-hand side graph R. The intermediate graph K is the context graph, which contains the nodes and edges in both, L and R, i.e. all nodes and edges matched to K remain unchanged during the transformation.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Fig. 3. Double-pushout approach

The application of a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ to G consists of transforming G into H by performing the construction of D and H such that (1) and (2) in Fig. 3 are pushouts. A more implementation-oriented interpretation of a rule application is that all nodes and edges in $m(L \setminus l(K))$ are removed from G to create $D = (G \setminus m(L)) \cup m(l(K))$ and then all nodes and edges in $n(R \setminus r(K))$ are added to create $H = D \cup n(R \setminus r(K))$.

Example 3. In Fig. 1 two graph production rules are shown in their shorthand notation. The numbers for the nodes imply the l and r morphisms and the graph K is implied as well and consists of the nodes with no edges for the first rule and a single edge of type a for the second rule. The first rule when applied to a graph G replaces an edge of type a by an edge of type b, whereas the second rule makes a similar replacement within a larger context.

2.2 Constraint Handling Rules (CHR)

This section presents the syntax and operational semantics of constraint handling rules [5]. Intuitively, CHR is a rule-based multiset rewriting system. For this work we consider a subset of CHR by considering only one kind of rule and omitting guards. For the complete possibilities of CHR see [4, 5].

The constraints manipulated by CHR are first-order predicates which we separate into *built-in constraints* and *user-defined constraints*. Built-in constraints are handled by a constraint solver while user-defined constraints are defined by a CHR program. In this work the required built-in constraints are syntactic equalities and basic arithmetics.

Simplification rules are of the form

$$\mathit{Rulename} \ @ \ H_1, \ldots, H_i \Leftrightarrow B_1, \ldots, B_k$$

where *Rulename* is an optional unique identifier of a rule, the head $H = H_1, \ldots, H_i$ is a non-empty conjunction of user-defined constraints, and the body $B = B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints. Note that we make sloppy use of the terms conjunction, sequence, and multiset with respect to $H_1, \ldots, H_i$ and $B_1, \ldots, B_k$.

The operational semantics is based on an underlying *constraint theory* $\mathcal{CT}$ for the built-in constraints and a *state*, which is a tuple $\langle G, C, \mathcal{V} \rangle$ where G is a goal store, i.e. a multiset of user-defined constraints, C is a conjunction of built-in constraints, and $\mathcal{V}$ is the set of global variables. The variables in $\mathcal{V}$ are also called variables-of-interest and, intuitively, differ from other variables occurring in the state by being considered universally quantified [5]. When comparing different states we make use of an equivalence relation $\equiv$ on CHR states. This equivalence accounts for different syntactical representations, including renaming of local variables, equality substitutions, and logically equivalent built-in stores.

A simplification rule of the form $H \Leftrightarrow B$ is *applicable* to a state $\langle E \cup G, C, \mathcal{V} \rangle$ if $CT \models \forall(C \rightarrow \exists \overline{x}(H = E))$ where $\overline{x}$ are the variables in H and $=$ is syntactic equality. The above condition intuitively corresponds to finding a subset E of constraints in the state that, together with the built-in store C, match the head of a rule $(H = E)$. We then define the following state transition for the application of the rule: $\langle E \cup G, C, \mathcal{V} \rangle \rightarrowtail \langle B_u \cup G, (H = E) \wedge C \wedge B_b, \mathcal{V} \rangle$ where $B = B_u \cup B_b$ with B_u being user-defined and B_b being built-in constraints. As usual, $\rightarrowtail^*$ denotes the reflexive-transitive closure of the $\rightarrowtail$ relation. When considering multiple programs $\rightarrowtail_{\mathcal{P}}$ denotes a transition based on a rule from program $\mathcal{P}$.

2.3 GTS in CHR

In this section we present how a graph transformation system can be embedded into CHR based on previous work in [3]. In order to embed a GTS in CHR, we have to encode its graph production rules as CHR rules and provide a conjunction of goal constraints corresponding to the host graph. We then recapitulate the soundness and completeness results of our embedding.

For encoding a GTS in CHR we first determine the constraints needed for encoding the rules and host graph based on the type graph:

Definition 3 (Type Graph Encoding).
For a type graph $\mathcal{TG}$ we define the set $\mathcal{C}$ of required constraints to encode graphs typed over $\mathcal{TG}$ as the minimal set including $v/2 \in \mathcal{C}$ for $v \in V_{\mathcal{TG}}$ and $e/3 \in \mathcal{C}$ for $e \in E_{\mathcal{TG}}$.

Example 4 (cont). For our example of the GTS, every node in the typed graph has the same type and we have two edge types. Based on this we need the following constraints: $\mathrm{n}/2, \mathrm{a}/3, \mathrm{b}/3$

We assume all nodes and edges of the type graph TG to be uniquely labeled such that the introduced constraints have unique names as well. Note that this is no restriction on the typed graphs as there can be any number of nodes or edges of the same type. These constraints allow us to encode typed graphs. The following definition of `chr` distinguishes between $\mathtt{chr}(\mathrm{host}, \cdot)$ and $\mathtt{chr}(\mathrm{keep}, \cdot)$ which intuitively correspond to host and rule graphs.

Definition 4 (Typed Graph Encoding).
For a typed graph G based on a type graph TG the set of constraints encoding G is defined differently for host and rule graphs. We define the following mappings for the encoding for an infinite set of variables VARS:

- $[\mathrm{type}_G(x)]$ *denotes the corresponding constraint name for encoding a node or edge of the given type.*
- $\mathrm{var} : G \to \mathit{VARS}, x \mapsto X_x$ *such that X_x is a unique variable associated to x, i.e.* var *is injective for the set of all graph nodes and edges.*
- $\mathrm{dvar} : G \to \mathit{VARS}, x \mapsto X_x$ *such that X_x is a unique variable associated to x, i.e.* dvar *is injective for the set of all graph nodes and edges.*

Using these mappings we define the following encoding of graphs:

$$\mathtt{chr}_G(\mathrm{host}, x) = \begin{cases} [\mathrm{type}_G(x)](\mathrm{var}(x), \mathrm{deg}_G(x)) & \text{if } x \in V_G \\ [\mathrm{type}_G(x)](\mathrm{var}(x), \mathrm{var}(\mathrm{src}(x)), \mathrm{var}(\mathrm{tgt}(x))) & \text{if } x \in E_G \end{cases}$$

$$\mathtt{chr}_G(\mathrm{keep}, x) = \begin{cases} [\mathrm{type}_G(x)](\mathrm{var}(x), \mathrm{dvar}(x)) & \text{if } x \in V_G \\ [\mathrm{type}_G(x)](\mathrm{var}(x), \mathrm{var}(\mathrm{src}(x)), \mathrm{var}(\mathrm{tgt}(x))) & \text{if } x \in E_G \end{cases}$$

We make use of the notations $\mathtt{chr}(\mathrm{host}, G) = \{\mathtt{chr}_G(\mathrm{host}, x) \mid x \in G\}$ *and* $\mathtt{chr}(\mathrm{keep}, G) = \{\mathtt{chr}_G(\mathrm{keep}, x) \mid x \in G\}$. *Furthermore, we omit the index G if the context is clear. For a node v encoded with* $\mathtt{chr}(\mathrm{keep}, v)$ *we call* $\mathrm{dvar}(v)$ *the* degree variable*.*

Example 5 (cont). When using the left-hand side graph of the first rule as a host graph G it is encoded in $\mathtt{chr}(\mathrm{host}, G)$ as follows:

$$\mathrm{n}(N_1, 1), \mathrm{n}(N_2, 1), \mathrm{a}(E_1, N_1, N_2)$$

The same graph G occurring as a rule graph is encoded in $\texttt{chr}(\mathrm{keep}, G)$ as follows:

$$\mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{a}(E_1, N_1, N_2)$$

We can now encode a complete graph production rule based on these definitions:

Definition 5 (GTS Rule in CHR).

For a graph production rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *from a GTS we define* $\rho(p) = (C_L, C_R)$ *with*

- $C_L = \{\texttt{chr}(\mathrm{keep}, x) \mid x \in K\} \cup \{\texttt{chr}(\mathrm{host}, x) \mid x \in L \setminus K\}$
- $C_R = \{\texttt{chr}(\mathrm{host}, x) \mid x \in R \setminus K\} \cup \{\texttt{chr}(\mathrm{keep}, e) \mid e \in E_K\}$ $\cup\{\texttt{chr}(\mathrm{keep}, v'), \mathrm{var}(v) = \mathrm{var}(v'), \mathrm{dvar}(v') = \mathrm{dvar}(v) - \deg_L(v) + \deg_R(v) \mid v \in V_K\}$

The rule p *is then encoded in CHR using* $\rho(p) = (C_L, C_R)$ *and in abuse of notation we use* $\rho(p)$ *for the CHR rule* p @ $C_L \Leftrightarrow C_R$ *as well as for the tuple* (C_L, C_R).

Example 6 (cont.). As an example, consider the first rule from our example GTS, which replaces the a-edge by a b-edge. Its encoding as a CHR simplification rule is given below:

$$\begin{aligned}\mathrm{r1}\ @\ & \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{a}(E_1, N_1, N_2)\\ & \Leftrightarrow\\ & \mathrm{n}(N_1', D_1'), N_1' = N_1, D_1' = D_1 - 1 + 1,\\ & \mathrm{n}(N_2', D_2'), N_2' = N_2, D_2' = D_2 - 1 + 1,\\ & \mathrm{b}(E_2, N_1, N_2)\end{aligned}$$

The above rule is created strictly according to Def. 5, but contains numerous superfluous constructs, like $D_1' = D_1 - 1 + 1$. Eliminating redundant expressions leads to the following simpler, yet equivalent, rule:

$$\begin{aligned}\mathrm{r1}\ @\ & \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{a}(E_1, N_1, N_2)\\ & \Leftrightarrow\\ & \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{b}(E_2, N_1, N_2)\end{aligned}$$

For the soundness and completeness results of this embedding we have to make sure, that the CHR programs only work on valid encodings of graphs. While in CHR any combination of node and edge constraints can be considered as input, not all of those make sense in terms of a GTS. To avoid such problems – like inconsistent degree encodings or dangling edges – we make use of the following graph invariant. This invariant holds for all states that are the valid encoding of a corresponding graph.

Definition 6 (Invariant, Graph Invariant).
$\mathcal{I}(S)$ is a property such that for all S_0 and S_1, we have that if $S_0 \rightarrow S_1$ (or $S_0 \equiv S_1$) and $\mathcal{I}(S_0)$ holds then $\mathcal{I}(S_1)$ holds.

The graph invariant *$\mathcal{G}(\sigma)$ with $\sigma = \langle E, C, \mathcal{V}\rangle$ holds if there exist a graph G and a conjunction of equality constraints C', such that*

$$\langle E, C \wedge C', \emptyset\rangle \equiv \langle \texttt{chr}(\text{host}, G), \top, \emptyset\rangle.$$

For a state σ where $\mathcal{G}(\sigma)$ holds with a graph G we say σ is a $\mathcal{G}$-state based on *G.*

Another characterization of σ being a $\mathcal{G}$-state based on G is the expression $\sigma \equiv \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$.

Using the notion of a $\mathcal{G}$-state based on G we can further identify the set of strong nodes. Those nodes are special to the CHR encoding of a GTS as the operational semantics of CHR ensures they cannot be deleted by rule applications. In [3] it is shown that they are the key to the strong joinability analysis of critical pairs, but we also need them here for the completeness result.

Definition 7 (Strong Nodes and Derivations).
For a CHR state $S = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ which is a $\mathcal{G}$-state based on G we define the set of strong nodes *as: $\mathcal{S}(S) = \{v \in V_G \mid \text{dvar}(v) = \text{deg}_G(v) \notin C\}$*

A GTS derivation $G \overset{p,m}{\Longrightarrow} G'$ using $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is strong with respect to *$S \subset V_G$ if $\forall s \in S : s \in m(K) \vee s \notin m(L)$.*

Finally, we present the soundness and completeness results from [3]. The soundness result states that CHR computations correspond to strong derivations:

Theorem 1 (Soundness).
Let $\rho(p) = (C_L, C_R)$ be a rule applicable to $\sigma = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ where $\mathcal{G}(\sigma)$ holds, such that $\sigma \rightarrowtail \sigma'$.

Then $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is applicable to G such that $G \overset{p,m}{\Longrightarrow} G'$ is strong w.r.t. $\mathcal{S}(\sigma)$. Furthermore, $\sigma' \equiv \langle \texttt{chr}(\text{keep}, G'), C', \mathcal{V}\rangle$ and $\mathcal{G}(\sigma')$ holds.

As mentioned previously the set of strong nodes cannot be deleted by CHR rule applications. Therefore, our completeness result is restricted such that only strong GTS derivations are possible in CHR. Note that this restriction becomes redundant if $\mathcal{S}(\sigma) = \emptyset$, which is especially true, when σ is a $\texttt{chr}(\text{host}, G)$-based encoding of a graph G.

Theorem 2 (Completeness).
Let $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, $G \overset{p,m}{\Longrightarrow} G'$, and let $\sigma = \langle \texttt{chr}(\text{keep}, G), C, \mathcal{V}\rangle$ be a $\mathcal{G}$-state based on G.

If $\forall x \in L \setminus K : m(x) \notin \mathcal{S}(\sigma)$, then $\rho(p) = (C_L, C_R)$ is applicable to σ. Furthermore, for $\sigma \rightarrowtail \sigma'$ it follows that $\sigma' \equiv \langle \texttt{chr}(\text{keep}, G'), C', \mathcal{V}\rangle$ and $\mathcal{G}(\sigma')$ holds.

2.4 Operational Equivalence in CHR

CHR is well-known for its decidable, sufficient, and necessary criterion for operational equivalence of terminating and confluent CHR programs [5, 6].

The understanding of operational equivalence within the CHR community intuitively means that the two programs should be able to compute equivalent outputs given the same input. Applied to a single state this behavior is called $\mathcal{P}_1, \mathcal{P}_2$-joinability:

Definition 8 ($\mathcal{P}_1, \mathcal{P}_2$-Joinable, Operational Equivalence).
Let $\mathcal{P}_1, \mathcal{P}_2$ be CHR programs. A state σ is $\mathcal{P}_1, \mathcal{P}_2$-joinable*, iff there are computations $\sigma \rightarrowtail^*_{\mathcal{P}_1} \sigma_1$ and $\sigma \rightarrowtail^*_{\mathcal{P}_2} \sigma_2$ with $\sigma_1 \equiv \sigma_2$ where all σ_i are final states with respect to $\mathcal{P}_i$.*
$\mathcal{P}_1, \mathcal{P}_2$ are operationally equivalent *iff all states σ are $\mathcal{P}_1, \mathcal{P}_2$-joinable.*

A decision algorithm for operational equivalence of CHR programs is presented in [6]. It is based on the analysis of critical states:

Definition 9 (Critical States).
Let $\mathcal{P}_1, \mathcal{P}_2$ be CHR programs. The set of critical states of $\mathcal{P}_1$ and $\mathcal{P}_2$ *is defined as* $\{\langle H, \top, \mathrm{vars}(H)\rangle \mid (H \Leftrightarrow B) \in \mathcal{P}_1 \cup \mathcal{P}_2\}$

Theorem 3 (Operational Equivalence via Critical States).
Let $\mathcal{P}_1, \mathcal{P}_2$ be terminating and confluent CHR programs. $\mathcal{P}_1, \mathcal{P}_2$ are operationally equivalent iff for all critical states σ of $\mathcal{P}_1$ and $\mathcal{P}_2$ it holds that σ is $\mathcal{P}_1, \mathcal{P}_2$-joinable.

3 Operational Equivalence of Graph Transformation Systems

In this section we introduce our notion of operational equivalence for GTS. Based on the embedding of GTS in CHR from [3] we use the existing decision algorithm from CHR for operational equivalence of two graph transformation systems.

As a basis we define the property of $\mathcal{S}_1, \mathcal{S}_2$-joinability for two graph transformation systems $\mathcal{S}_1, \mathcal{S}_2$. It is motivated by $\mathcal{P}_1, \mathcal{P}_2$-joinability in the context of CHR as given in Def. 8.

Definition 10 ($\mathcal{S}_1, \mathcal{S}_2$-joinability).
*Let $\mathcal{S}_1, \mathcal{S}_2$ be two graph transformation systems. A typed graph G is $\mathcal{S}_1, \mathcal{S}_2$-*joinable *iff there are derivations $G \Rightarrow^*_{\mathcal{S}_1} G_1$ and $G \Rightarrow^*_{\mathcal{S}_2} G_2$ with $G_1 \simeq G_2$ being final w.r.t. $\mathcal{S}_1$ and $\mathcal{S}_2$. Here $\simeq$ denotes traditional graph isomorphism and a graph G is considered* final *w.r.t. $\mathcal{S}$ if there is no transition $G \Rightarrow_{\mathcal{S}} H$ for any graph H.*

Building on $\mathcal{S}_1, \mathcal{S}_2$-joinability we can now define operational equivalence for graph transformation systems with the same intuitive understanding that two equivalent GTS should be able to produce the same result graphs up to isomorphism given an input graph:

Definition 11 (GTS Operational Equivalence).
Let $\mathcal{S}_1 = (\mathcal{P}_1, \mathcal{TG})$ and $\mathcal{S}_2 = (\mathcal{P}_2, \mathcal{TG})$ be two graph transformation systems. $\mathcal{S}_1, \mathcal{S}_2$ are operationally equivalent *iff for all graphs G typed over $\mathcal{TG}$ it holds that G is $\mathcal{S}_1, \mathcal{S}_2$-joinable.*

Similar to operational equivalence in CHR where it is futile to directly compare programs that use different constraints, Def. 11 requires $\mathcal{S}_1$ and $\mathcal{S}_2$ to be based on the same type graph $\mathcal{TG}$. With the previous results from [3] we can directly use CHR's operational equivalence as a sufficient criterion for deciding operational equivalence of two GTS:

Theorem 4 (CHR Operational Equivalence Implies GTS Operational Equivalence).
Let $\mathcal{S}_1, \mathcal{S}_2$ be graph transformation systems and $\mathcal{P}_1, \mathcal{P}_2$ their corresponding CHR programs. $\mathcal{S}_1, \mathcal{S}_2$ are operationally equivalent if $\mathcal{P}_1, \mathcal{P}_2$ are operationally equivalent.

Proof. *Let G be a graph typed over $\mathcal{TG}$. Then the state $\sigma = \langle \texttt{chr}(\text{host}, G), \top, \emptyset \rangle$ is $\mathcal{P}_1, \mathcal{P}_2$-joinable by Def. 8. Therefore, there exist the final states $\sigma_1 \equiv \sigma_2$ with $\sigma \rightarrowtail^*_{\mathcal{P}_1} \sigma_1$ and $\sigma \rightarrowtail^*_{\mathcal{P}_2} \sigma_2$. By Thm. 1 we know that there exist corresponding derivations $G \Rightarrow^*_{\mathcal{S}_1} G_1$ and $G \Rightarrow^*_{\mathcal{S}_2} G_2$ such that σ_1 is a $\mathcal{G}$-state based on G_1 and σ_2 is a $\mathcal{G}$-state based on G_2. Due to Thm. 2 G_1 and G_2 are final states w.r.t. $\mathcal{S}_1$ and $\mathcal{S}_2$, and finally, the isomorphism between G_1 and G_2 is implied by $\sigma_1 \equiv \sigma_2$. Therefore, G is $\mathcal{S}_1, \mathcal{S}_2$-joinable.* □

The reverse direction of Thm. 4 cannot be proved in a similarly simple way. This is due to the problem, that $\mathcal{P}_1, \mathcal{P}_2$-joinability is required for all states, including states that have no corresponding graph. By restricting observation to valid states we plan to derive a stronger characterization of operational equivalence of GTS (see Sect. 4).

3.1 Redundant Rule Removal

The redundant rule removal algorithm is an application of operational equivalence presented in [11]. It can be applied to graph transformation systems embedded in CHR. It requires the CHR program to be terminating and confluent with respect to states corresponding to graphs. This implies that the GTS is required to be confluent and terminating as well [3]. Any redundant rule of such a program then corresponds to a redundant rule for derivations of the GTS.

The basic idea of the algorithm is to try to remove a rule from the program and compare this modified program with the original program. If both are operationally equivalent the rule is redundant and can be removed. Note that depending on the order in which rules are tried different results are possible and this algorithm, hence does not guarantee to yield an operationally equivalent program with the minimal number of rules possible. Nevertheless, it proved to be an important algorithm especially in the context of automatically generated programs [7, 8].

Example 7. Consider again the GTS given in Fig. 1 and its embedding in CHR:

$$\begin{array}{ll} \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), & \Leftrightarrow \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \\ \mathrm{a}(E, N_1, N_2) & \quad \mathrm{b}(E', N_1, N_2) \end{array}$$

$$\begin{array}{ll} \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{n}(N_3, D_3), & \Leftrightarrow \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{n}(N_3, D_3), \\ \mathrm{a}(E_1, N_1, N_2), \mathrm{a}(E_2, N_1, N_3) & \quad \mathrm{b}(E_1', N_1, N_2), \mathrm{a}(E_2, N_1, N_3) \end{array}$$

The algorithm tries to remove the second rule from the program and then compare the two programs according to the operational equivalence test. The only relevant case is considering the head of the removed rule as input to both programs. The two computations may use different rules, but both programs compute a final state consisting of a graph with three nodes, an a-edge from the first to the third node, and an a-edge from the first to the second node.

The previous example shows that our approach uses strong derivations for testing operational equivalence. This is an important feature and as the following example demonstrates implementing our approach directly in GTS, that use non-strong derivations, is not possible.

Example 8. Consider the two graph transformation systems shown in Fig. 4. Using the graph on the left-hand sides of the two rules as input to both GTSs leads to isomorphic results as shown in Fig. 5 (1). However, assuming the slightly extended graph shown in Fig. 5 (2) instead gives two non-isomorphic result graphs. Therefore, only examining the critical states within the GTS using non-strong derivations is insufficient. Applying our approach to the GTS in Fig. 4, however, gives the intended result. We assume this difference is due to the isomorphism function and the track morphisms (see [12]) for the two GTS derivations being inconsistent (see also Sect. 4).

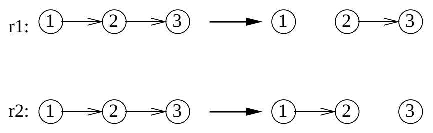

Fig. 4. Two operationally non-equivalent graph transformation systems

4 Conclusion

In this work we introduced operational equivalence of graph transformation systems. The proposed notion of operational equivalence is motivated by research in the context of CHR based on joinability of states. Interestingly, our previous work on embedding GTS in CHR [3] provides the means to directly derive

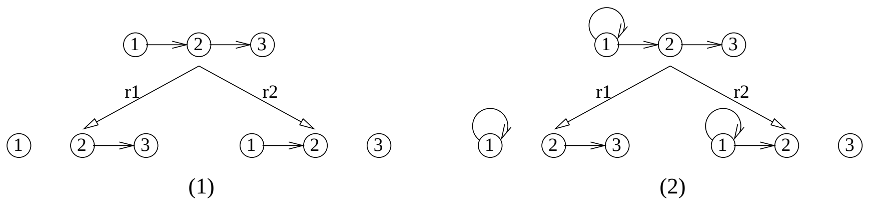

Fig. 5. Derivations for two initial graphs

a sufficient criterion for operational equivalence of two graph transformation systems.

As a direct application of this work we showed that our criterion is suitable for automatically removing redundant rules from a GTS. Finally, we have given an example demonstrating that a direct translation of our approach into the GTS context does not yield the expected outcome.

4.1 Future Work

As the final example showed for a GTS embedded in CHR nodes are implicitly tracked. We exploited this behavior earlier in [3] in order to decide strong joinability which in the GTS context is formulated via an explicit track morphism [12]. We hope to find a similar formulation of the operational equivalence test, in which an explicit check involving the track morphism allows us to reach a corresponding sufficient criterion within the GTS context.

The redundant rule used in the example in Fig. 1 in this work is *subsumed* by the other rule. This notion of subsumption for graph transformation systems is discussed by Kreowski and Valiente in [13] and an interesting future line of research is to compare our approach with that work. Because our approach relies on termination it cannot be as generic as their sufficient condition. However, to the best of our knowledge and as stated in [13] it remains open to find a verification procedure for that condition. Our approach might, hence, be able to verify redundancy for a subset of the rules that are redundant according to the criterion of Kreowski and Valiente.

Furthermore, in the context of CHR the notion of *operational c-equivalence* for a constraint c was introduced to extend the classes of programs that can be operationally equivalent. This notion is not suitable for the context of GTS as it corresponds to two graph transformation systems defined over two type graphs that share exactly one node. A more realistic case is the sharing of a subgraph of the type graphs which in turn gives rise to the idea of *operational C-equivalence* for a set of constraints C. Another reason to examine this also in the context of CHR is that a notion of operational C-equivalence is a generalization of both previous equivalence notions.

References

1. Blostein, D., Fahmy, H., Grbavec, A.: Practical use of graph rewriting. In: 5th Workshop on Graph Grammars and Their Application To Computer Science. Volume 1073 of Lecture Notes in Computer Science., Springer-Verlag (1995) 38–55
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag (2006)
3. Raiser, F., Frühwirth, T.: Strong joinability analysis for graph transformation systems in CHR. In: 5th International Workshop on Computing with Terms and Graphs, TERMGRAPH'09. (2009)
4. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Accepted to *Journal of Theory and Practice of Logic Programming* (2009)
5. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009) to appear.
6. Abdennadher, S., Frühwirth, T.: Operational equivalence of CHR programs and constraints. In Jaffar, J., ed.: Principles and Practice of Constraint Programming, CP 1999. Volume 1713 of Lecture Notes in Computer Science., Springer-Verlag (1999) 43–57
7. Raiser, F.: Semi-automatic generation of CHR solvers for global constraints. In Stuckey, P.J., ed.: Principles and Practice of Constraint Programming, 14th International Conference, CP 2008. Volume 5202 of Lecture Notes in Computer Science., Sydney, Australia, Springer-Verlag (September 2008) 588–592
8. Abdennadher, S., Sobhi, I.: Generation of rule-based constraint solvers: Combined approach. In King, A., ed.: Logic-Based Program Synthesis and Transformation, 17th International Symposium, LOPSTR 2007, Kongens Lyngby, Denmark, August 23-24, 2007, Revised Selected Papers. Volume 4915 of Lecture Notes in Computer Science., Springer-Verlag (2007) 106–120
9. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In: Proc. of ICGT '08 (International Conference on Graph Transformation). Volume 5214 of Lecture Notes in Computer Science., Springer-Verlag (2008) 242–256
10. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In Walukiewicz, I., ed.: Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004. Volume 2987 of Lecture Notes in Computer Science., Barcelona, Spain, Springer-Verlag (2004) 151–166
11. Abdennadher, S., Frühwirth, T.: Integration and optimization of rule-based constraint solvers. In Bruynooghe, M., ed.: Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers. Volume 3018 of Lecture Notes in Computer Science., Springer-Verlag (2003) 198–213
12. Plump, D.: Confluence of graph transformation revisited. In Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R.C., eds.: Processes, Terms and Cycles. Volume 3838 of Lecture Notes in Computer Science., Springer-Verlag (2005) 280–308
13. Kreowski, H.J., Valiente, G.: Redundancy and subsumption in high-level replacement systems. In: TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations, Springer-Verlag (2000) 215–227