Graph Transformation Systems in CHR

Frank Raiser

Faculty of Engineering and Computer Sciences, University of Ulm, Germany
Frank.Raiser@uni-ulm.de

Abstract. In this paper we show it is possible to embed graph transformation systems (GTS) soundly and completely in constraint handling rules (CHR). We suggest an encoding for the graph production rules and we investigate its soundness and completeness by ensuring equivalence of rule applicability and results. We furthermore compare the notion of confluence in both systems and show how to adjust a standard CHR confluence check to work for an embedded GTS.

1 Introduction

Constraint handling rules (CHR) [1] allow for rapid prototyping of constraint-based algorithms. Besides constraint reasoning, CHR have been used for various tasks including theorem proving, parsing, and multi-set rewriting. In this work we show that CHR also provide the means for concise implementations of graph transformations.

Graph transformation systems are used to describe complex structures and systems in a concise, readable, and easily understandable way. They have applications ranging from implementations of programming languages over model transformations to graph based models of computation [2–4]. The principal idea of a graph transformation is to apply a production rule to a graph. A part of the so-called host graph has to be matched to the rule and is then modified accordingly.

While there are several specialized tools available for performing graph transformations, it is usually cumbersome to combine them with general-purpose programming languages. Following the tradition of using CHR for rapid prototyping, we investigate the necessities for implementing a graph transformation system with CHR. We show that CHR are indeed a suitable choice for prototyping graph transformations, as every rule of a graph transformation system can be directly and intuitively translated into a corresponding CHR rule already yielding an executable graph transformation system. No effort has to be invested into creating an underlying transformation engine, as the CHR implementation in combination with an advantageous encoding of the rules is sufficient.

In order to arrive at an intuitive and concise encoding of graph production rules with CHR, we make restrictions to the graph transformation systems. These restrictions consist in requiring injective matchings and inclusions in the graph production rules. However, these are common restrictions often found in practical applications of graph transformation systems [4, 5].

From a theoretical point of view we investigate the soundness and completeness of our proposed encoding by ensuring equivalence of applicability and results. Thus we ensure that such a CHR rule is applicable if and only if the corresponding graph production rule is applicable. We further show that applying a constraint handling rule in this context results in a state encoding the graph obtained by applying the corresponding graph production rule in a graph transformation system and vice versa.

Finally, we compare the notion of confluence in both systems. We particularly concentrate on the investigation of critical pairs which is the basis of confluence checking. A slightly changed definition of CHR confluence is presented that allows the application of existing confluence checkers to a graph transformation system embedded in CHR.

This work is divided into six sections: We begin with the introduction of the necessary notions of graph transformation systems and CHR in Section 2. Section 3 then presents our proposed encoding of a GTS in CHR, the properties of which we analyze in Section 4. Section 5 compares the notion of confluence in both systems before we conclude in Section 6.

2 Preliminaries

In this section we introduce the required formalisms for graph transformation systems and constraint handling rules.

2.1 Graph Transformation System (GTS)

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 set V of nodes, a 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 the purpose of simplicity, we avoid an additional label morphism in favor of identifying variable names with their labels. 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* $\deg : V \to \mathbb{N}, v \mapsto \#\{e \in E \mid \mathrm{src}(e) = v\} + \#\{e \in E \mid \mathrm{tgt}(e) = v\}$.

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* $\mathrm{type}_V : V \to TG_V, \mathrm{type}_E : E \to TG_E$. *The* type *morphism is a graph morphism, therefore it has to satisfy the following condition:* $\forall e \in E : \mathrm{type}_V(\mathrm{src}(e)) = \mathrm{src}_{TG}(\mathrm{type}_E(e)) \wedge \mathrm{type}_V(\mathrm{tgt}(e)) = \mathrm{tgt}_{TG}(\mathrm{type}_E(e))$

Example 1. Figure 1 shows an example for a type graph and a corresponding typed graph. The type graph is used to define two types of nodes: processes and resources. Furthermore it allows *use* edges going from processes to resources.

The typed graph is one possible instance of a graph modelling processes and resources being used by those processes. The type morphism is represented by the dotted lines, showing how the nodes are typed as processes or resources, respectively.

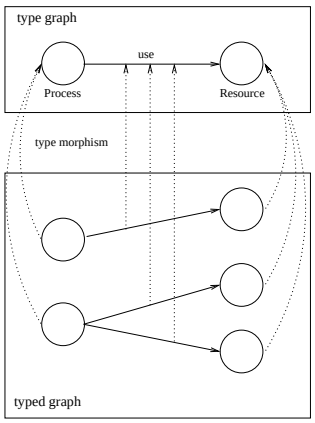

Fig. 1. Type graph and typed graph

Definition 2 (GTS, rule). *A* Graph Transformation System *(GTS) is a tuple consisting of a type graph and a 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 with inclusion morphisms* $l : K \to L$ *and* $r : K \to R$.

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 as defined in [2]. Most notably, we require a so-called match morphism $m : L \to G$ to apply a rule p to a typed host graph G. The transformation yielding the typed graph H is written as $G \overset{p,m}{\Longrightarrow} H$. For our work, we restrict the possible match morphisms to injective morphisms which is a common restriction [4, 5].

A graph production rule p can only be applied to a host graph G if the gluing condition [2] is satisfied, which in our case of injective match morphisms is simplified such that p can be applied as long as there are no dangling edges. Section 3.1 explains the notion of dangling edges and how they are handled in our encoding.

As an example, we provide an implementation for recognizing cyclic lists, which is presented in [4].

Example 2. The GTS for recognizing cyclic lists consists of two rules depicted in Figure 2. The rule *unlink* shortens a list consisting of three linearly connected

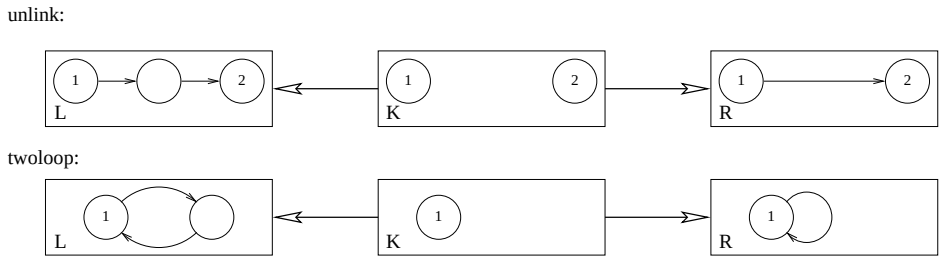

Fig. 2. Graph transformation system for recognizing cyclic lists

nodes by removing the intermediate node. For a cyclic list, this rule can be applied up to the point where only two nodes are left. In that case, the rule *twoloop* transforms those two nodes into the graph consisting of a single node with a loop. See [4] for a more thorough discussion of this example.

Note that we use a shorthand notation in Figure 2 which only shows the morphisms l and r implicitly by the labels of the nodes which are mapped onto each other. Nodes and edges which are removed or added in the graphs L or R are not labelled, as there is no node or edge in K which is mapped to them.

2.2 Constraint Handling Rules (CHR)

This section presents the syntax and operational semantics of Constraint Handling Rules [1, 6]. Constraints are first-order predicates which we separate into *built-in constraints* and *user-defined constraints*. Built-in constraints are provided by the constraint solver while user-defined constraints are defined by a CHR program. For our purpose we only need a subset of CHR, namely *simplification rules*. Simplification rules are of the form

$$\mathit{Rulename} \ @ \ H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k$$

where *Rulename* is a unique identifier of a rule, the head $H = H_1, \ldots, H_i$ is a non-empty conjunction of user-defined constraints, the guard $\overline{G} = G_1, \ldots, G_j$ is a conjunction of built-in constraints and the body $B = B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints.

The operational semantics of a simplification rule is based on an underlying constraint theory CT for the built-in constraints and a state, which is a pair $\langle G, C\rangle$ where G is a goal, i.e. a conjunction of user-defined and built-in constraints, and C is a (built-in) constraint [6]. A simplification rule of the form $H \Leftrightarrow \overline{G} \mid B$ is applicable to a state $\langle E \wedge G, C\rangle$ if $CT \models \forall(C \rightarrow \exists\overline{x}(H \doteq E \wedge \overline{G}))$ where $\overline{x}$ are the variables in H. We define the following state transition for the application: $\langle E \wedge G, C\rangle \mapsto \langle B \wedge G, (H \doteq E) \wedge C \wedge \overline{G}\rangle$.

For the remainder of this work, we can further restrict these rules, as there is no need for the guard constraints, such that a CHR simplification rule is simply of the form

$$\mathit{Rulename} \ @ \ H \Leftrightarrow B.$$

3 Representation of Graphs in CHR

In order to embed a GTS in CHR, we have to encode the available graph production rules as simplification rules and provide a conjunction of goal constraints corresponding to the host graph. To this end we provide a bijective correspondence between graphs and their representation through CHR constraints given by the following constructions.

At first we have to find out what constraints will be needed for encoding the rules and host graph. At this point we require the GTS to be typed, as we can directly defer the necessary constraints from the corresponding type graph as explained in Definition 3. Note that this is a very weak restriction though, as every untyped graph can be typed over the trivial type graph consisting of a single node with a loop.

Definition 3 (type graph encoding). *For a type graph TG we define the following constraints to encode graphs typed over TG:*

- *Introduce a constraint* $\text{degree}/2$.
- $\forall v \in V_{TG}$ *introduce a constraint* $v/1$.
- $\forall e \in E_{TG}$ *introduce a constraint* $e/3$.

We assume all nodes and edges of the type graph TG to be uniquely labelled such that the introduced constraints have unique names as well. The $\text{degree}/2$ constraint is a special constraint we require to be able to check for dangling edges as is explained in Section 3.1.

Definition 4 (typed graph encoding). *A typed graph G based on a type graph TG is encoded with constraints as follows:*

- $\forall v \in V_G$ *with* $\text{type}(v) = t$ *add the constraint* $t(T)$ *with* T *being a new variable.*
- $\forall e \in E_G$ *with* $\text{type}(e) = t, \text{src}(e) = v_1, \text{tgt}(e) = v_2, \text{type}(v_1) = t', \text{type}(v_2) = t''$ *and previously created constraints* $t'(T^1), t''(T^2)$ *add another constraint* $t(E, T^1, T^2)$ *with* E *being a new variable.*
- *If G is a typed host graph the node degrees are known and thus* $\forall v \in V_G$ *with* $deg(v) = k, \text{type}(v) = t$ *which add* $t(T)$ *we also add* $\text{degree}(T, k)$.

A typed rule graph G encoded like this is denoted as $\text{encode}(G)$.

Example 3 (cont). For our example of the GTS for recognizing cyclic lists we assume the trivial type graph consisting only of a node and a loop. Therefore every node in the typed graph has the same type, just like every edge has the same type. Based on this type graph we need the following constraints: $\text{degree}/2, \text{node}/1, \text{edge}/3$.

A host graph which contains a simple cyclic list consisting of exactly two nodes is encoded as follows:
$\text{node}(N_1) \wedge \text{node}(N_2) \wedge \text{edge}(E_1, N_1, N_2) \wedge \text{edge}(E_2, N_2, N_1) \wedge \text{degree}(N_1, 2) \wedge \text{degree}(N_2, 2)$.

We can now encode a complete graph production rule based on these definitions. There are several possible ways to do this in CHR. However, we restrict ourselves to a single simplification rule here, as it is a very intuitive encoding which is useful in the upcoming proofs. The remaining results of this paper can also be transferred to different CHR representations like a simpagation rule approach.

Definition 5 (GTS rule in CHR). *A graph production rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *from a GTS is translated into a CHR simplification rule of the form* $p \mathbin{@} H \Leftrightarrow B$ *with* H, B *being conjunctions of constraints as follows:*

- $H = \text{encode}(L) \wedge D_1$
- $B = \text{encode}(R) \wedge D_2$
- D_1 *and* D_2 *being conjunctions of* degree *constraints as follows:*
  - $\forall v \in L \setminus K$ *with* $\deg(v) = k$, $\text{type}(v) = t$, *and* $t(T) \in \text{encode}(L)$ *we have* $\text{degree}(T, k) \in D_1$.
  - $\forall v \in K$ *let* $n = \#\{e \in E_L \mid \text{src}(e) = v \vee \text{tgt}(e) = v\}$, $m = \#\{e \in E_R \mid \text{src}(e) = v \vee \text{tgt}(e) = v\}, type(v) = t, t(T) \in \text{encode}(K)$ *then* $\text{degree}(T, D) \in D_1$ *with* D *being an unused variable. Further add the constraint* $\text{degree}(T, D{+}m{-}n) \in D_2$.
  - $\forall v \in R \setminus K$ *with* $\deg(v) = k, type(v) = t, t(T) \in C_2$ *let* $\text{degree}(T, k) \in D_2$.

We further require the encoding of K *found in* $\text{encode}(L)$ *and* $\text{encode}(R)$ *to share the same variables for the same nodes and edges as this implicitly models the inclusion morphisms analogously to the way we use node labels in Figure 2.*

A rule encoded like this is denoted as $\text{code}(p)$.

Example 4 (cont). As an example, consider the second rule from our example GTS, which reduces two cyclic nodes to a single node with a loop. Its encoding as a CHR simplification rule is given below:

$$
\begin{aligned}
\text{twoloop} \mathbin{@}\ & \text{node}(N_1), \text{node}(N_2), \\
& \text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_1), \\
& \text{degree}(N_2, 2), \text{degree}(N_1, D_1) \\
& \Leftrightarrow \\
& \text{node}(N_1), \text{edge}(E, N_1, N_1), \\
& \text{degree}(N_1, D_1{+}2{-}2).
\end{aligned}
$$

3.1 On Dangling Edges

A graph production rule can only be applied to a host graph when it is guaranteed that the result is consistent. While in the most general case of graph transformations there also exists an identification problem [2], our injective matchings allow for only one kind of inconsistency: When a node gets deleted by a rule, the corresponding node in the host graph may have edges adjacent to it which are not explicitly given in the rule. In such a case, the remaining edge would be left *dangling* as it is no longer adjacent to two nodes. Therefore this situation has to be avoided and before a rule is applied to a host graph, we first have to ensure that there are no dangling edges according to the following definition:

Definition 6 (dangling edge). *A* dangling edge *is an edge* $e \in E_G \setminus m(E_L)$ *such that there is a node* $v \in V_L \setminus V_K$ *with* $m(v) = \mathrm{src}(e) \vee m(v) = \mathrm{tgt}(e)$.

The degree /2 constraints introduced earlier are our means of detecting dangling edges: Let $e \in E_G$ be a dangling edge which is adjacent to $v_G \in V_G$, such that for $v \in V_L \setminus V_K : m(v) = v_G$. Due to Definition 5, the corresponding rule includes a $\mathrm{degree}(T, k)$ constraint for the node v with $k = \deg(v)$. This means that there are k edges adjacent to the node v in the rule graph. When matching this rule graph injectively to the host graph G we need to identify each of these edges with an edge in E_G adjacent to $m(v) = v_G$. By the definition of a dangling edge, the edge e is not among those k edges as $e \in E_G \setminus m(E_L)$. Therefore, we have $\deg(v_G) > k$. As G is a host graph and thus the degrees of nodes are known, there is a corresponding $\mathrm{degree}(T', l)$ constraint with $l > k$ for the node v_G. Here it can be seen clearly that a match between $\mathrm{degree}(T', l)$ and $\mathrm{degree}(T, k)$ is impossible due to $l > k$ and therefore the rule will not be applied as the dangling edge condition is violated.

Note that this check is only relevant to nodes which get removed by the graph production rule. If a node is in V_K, a dangling edge is not possible and thus for those nodes the rule contains $\mathrm{degree}(T, D)$ constraints which can always be matched due to D being a variable.

Example 5 (cont). Consider the *twoloop* rule given in Example 4 along with the following encoded host graph:
$\mathrm{node}(A) \wedge \mathrm{node}(B) \wedge \mathrm{node}(C) \wedge \mathrm{edge}(E_1, A, B) \wedge \mathrm{edge}(E_2, B, A) \wedge \mathrm{edge}(E_3, B, C) \wedge \mathrm{degree}(A, 2) \wedge \mathrm{degree}(B, 3) \wedge \mathrm{degree}(C, 1)$
Applying the *twoloop* rule to this graph to remove the node B would leave the edge E_3 going from B to C dangling. However, this is avoided as the encoding of the *twoloop* rule contains the following constraint in its head: $\mathrm{degree}(N_2, 2)$. Only a node with a degree of exactly 2 can thus be removed by this rule, which rules out that the node B in the above host graph is removed. Nevertheless, the rule can be applied with $N_2 \doteq A$ as the node A has the necessary degree of 2.

3.2 Runtime Performance

After embedding graph transformations in CHR we are interested in the runtime performance needed to execute these transformations. In general the perfomance of a CHR program is determined by the time needed to find an applicable rule and the time needed to apply that rule.

As we have a one-to-one correspondence of CHR rules and graph transformation rules we know that executing D graph transformation steps is equal to applying D CHR rules in the corresponding CHR program. Given a host graph $G = (V, E, src, tgt)$ its encoding in CHR corresponding to Definition 4 uses $2|V| + |E|$ constraints. In general the search for an applicable rule needs to consider all possible combinations of these constraints for all constraints appearing in the head of a rule. Using the upper bound from [7] we get the runtime complexity as $\mathcal{O}(D \cdot \sum_i c_{max}^{n_i} \cdot (O_{H_i} + O_{G_i}) + (O_{C_i} + O_{B_i}))$ where the sum iterates

over all rules of the program, c_{max} is the maximal number of constraints present at any time during the computation, n_i is the number of head constraints in rule i, O_{H_i} is the time needed to unify the chosen goal constraints with the head constraints, O_{G_i} is the time needed to verify the guard of the rule, O_{C_i} is the time needed for adding the constraints on the right-hand side of the rule, and O_{B_i} is the time needed to remove the constraints on the left-hand side of the rule.

Note that in our embedding, CHR rules corresponding to graph transformation rules require no guards and thus $O_{G_i} = 0 \forall i$. Furthermore the time needed to add and remove constraints can be considered constant. For a worst-case analysis we can furthermore replace n_i with the $n = n_j$ of the rule j with the maximal number of head constraints. This yields a runtime complexity of $\mathcal{O}(D \cdot \sum_i c_{max}^n \cdot O_{H_i})$. The time required for unification with the head constraints can also be approximated by the rule with the largest number of head constraints. With m being the number of rules in the program we then get the complexity $\mathcal{O}(D \cdot m \cdot c_{max}^n \cdot O_{H_n})$.

Better bounds can be deferred by taking a closer look at the embedded GTS. For example if all rules have the property of not extending the graph's size, i.e. rules remove more edges and nodes than adding them, then the number of constraints is bounded by the number c_0 of initial constraints which encode the original host graph. Therefore the bound for such a GTS is $\mathcal{O}(D \cdot m \cdot c_0^n \cdot O_{H_n})$. Also note that actual CHR implementations use sophisticated methods to determine constraints for rule applications and thus the actual runtime may be considerably less.

Example 6. Consider the example GTS for recognizing cyclic lists from Figure 2. As it contains two rules which both remove a node and reduce the number of edges by one it is clear that the number of goal constraints monotonly decreases during execution. Therefore $c_{max} = c_0 = 2|V| + |E|$, i.e. the maximal number of constraints in a computation is bounded by the original host graph's encoding. Furthermore the number of rule applications is restricted by the number of nodes in the host graph, such that $D = |V|$. And as the two CHR rules are known we get $m = 2$ and $n = \max_i n_i = \max\{8, 6\} = 8$. Therefore runtime complexity is bounded by: $\mathcal{O}(|V| \cdot m \cdot (c_0)^8 \cdot O_{H_1})$. As $c_0 = 2|V| + |E|$ can be approximated by $|V|^2$ we get $\mathcal{O}(2|V| \cdot (|V|^2)^8 \cdot O_{H_1}) = \mathcal{O}(|V|^{17} \cdot O_{H_1})$. Note that this is a crude approximation and in practical settings runtime is much better depending on the chosen CHR implementation.

4 Soundness and Completeness

After introducing our encoding for a GTS in CHR, we now investigate the properties of this encoding. First of all, we show that the CHR rules created for a graph production rule are applicable if and only if the corresponding graph production rule is applicable. For a GTS one of the applicable rules is chosen

nondeterministically. We get this exact behavior when using the standard semantics of CHR.

Lemma 1 (CHR applicability). *If* $\mathrm{code}(p)$ *is applicable, so is the corresponding graph production rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$.

Proof. If the CHR rule is applicable, then there exist matching constraints for H *in the goal store. As no functions are used for the constraints in the goal store the matching only requires substitutions of the form* $[T/c]$ *or* $[T/T']$ *for variables* T, T' *and constant* c.

This CHR matching implies a match morphism m *for the graph production rule: Every node* $v \in V_L$ *(resp. edge* $e \in E_L$*) is represented as a constraint in* H *and there is a matching constraint* C' *which corresponds to a node* $v' \in V_G$ *(resp. edge* $e' \in E_G$*) such that we can define* $m(v) = v'$ *(resp.* $m(e) = e'$*). Due to the multi-set semantics of CHR, different nodes in* V_G *are encoded by different constraints and no single constraint can be matched to multiple head constraints. This property of CHR guarantees that* m *is injective.*

It remains to be shown that there are no dangling edges. Let us therefore assume that there is a dangling edge, i.e. there exists an edge $e \in E_G \setminus m(E_L)$ *with* $\mathrm{src}(e) = m(v) \vee \mathrm{tgt}(e) = m(v)$ *for a* $v \in L \setminus K$.

As $v \in L \setminus K$*, a constraint* $\mathrm{degree}(T, k)$ *is present in* H *according to Def. 5. Due to the CHR rule being applicable, this requires a matching constraint in the goal store. Therefore the node* $m(v)$ *has degree* k *as well. Due to* m *being an injective graph morphism, however, every edge adjacent to* v *is mapped to an edge adjacent to* $m(v)$*. Therefore the dangling edge* e *cannot exist.*

□

Lemma 2 (graph rule applicability). *If* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *is applicable, so is* $\mathrm{code}(p)$.

Proof. Let m *be the injective match morphism for the application of the graph production rule. Then* $\forall v \in V_L$ *with* $\mathrm{type}(v) = t$ *there is a constraint* $t(T)$ *in* H*. As* m *is a graph morphism, we have that* $\mathrm{type}(m(v)) = t$ *and thus a constraint* $t(T')$ *exists in the goal store corresponding to the node* $m(v)$*. The CHR rule is therefore applicable through a match with* $[T/T']$.

$\forall e \in E_L$ *with* $\mathrm{src}(e) = v_s, \mathrm{tgt}(e) = v_t, \mathrm{type}(e) = t_e, \mathrm{type}(v_s) = t_s, \mathrm{type}(v_t) = t_t$ *we have the following CHR rules in the conjunction* H*:* $t_s(T_1)$*,* $t_t(T_2)$*, and* $t_e(E, T_1, T_2)$*. Due to* m *being a graph morphism, there exist corresponding constraints* $t_s(T_1')$*,* $t_t(T_2')$ *for the nodes* $v_1 = m(v_s), v_2 = m(v_t)$ *in the goal. There further exists an edge* e' *with* $\mathrm{src}(e') = m(v_s), \mathrm{tgt}(e') = m(v_t), \mathrm{type}(e') = t_e$ *which is represented by a constraint* $t_e(E', T_1', T_2')$*. As discussed above, we already have* $[T_1/T_1'][T_2/T_2']$*, so we can extend this to* $[T_1/T_1'][T_2/T_2'][E/E']$ *for a CHR match.*

It remains to be shown that there is a match for the eventually occurring $\mathrm{degree}/2$ *constraints in* H*:*
Case 1: a constraint of the form $degree(T, D)$ *with* $[T/T']$ *is always matchable, as the* degree *constraint corresponding to the node represented by* T' *is guaranteed*

to be available in the goal store.
Case 2: $\text{degree}(T, k)$ *with* $[T/T']$*: Analogous to case 1, there exists a constraint* $\text{degree}(T', l)$ *in the goal store.* $k > l$ *is a contradiction to* m *being an injective graph morphism.* $k < l$ *implies an edge* $e' \notin m(E_L)$*. However constraints of the form* $\text{degree}(T, k)$ *are added only for nodes which are removed by the production rule and thus this is a dangling edge which contradicts the initial applicability of the graph production rule. Therefore* $k = l$*, which allows a CHR matching through* $[T/T']$*.*

□

Theorem 1 (applicability). *A graph production rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *is applicable to a typed host graph* G *if and only if* $\text{code}(p)$ *is applicable to* $\text{encode}(G)$*.*

Proof. This is immediate from the combination of Lemma 1 and Lemma 2.

□

Theorem 2 (soundness and completeness). *Given a typed host graph* G *and* $\text{encode}(G)$*, a graph production rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *represented by* $\text{code}(p) = H \Leftrightarrow B$*, and a match* $m : L \to G$ *the following transitions are equivalent:*

- $G \overset{p,m}{\Longrightarrow} \tilde{G}$
- $\langle \text{encode}(G), true \rangle \;\mapsto\; \langle \text{encode}(\tilde{G}), true \rangle$ *by applying* $\text{code}(p)$*.*

Proof. "⇒": According to the proof of Lemma 2, the match m *implies a match for applying* $\text{code}(p)$*. We now show that the construction of* $G \overset{p,m}{\Rightarrow} \tilde{G}$ *is analogous to the application of the CHR rule* $\text{code}(p)$*:*
First the nodes and edges in $m(L)$ *get deleted from* G*, but nodes and edges in* $m(l(K)) = m(K)$ *(*l *is an inclusion) are kept. For the CHR rule application, all constraints in the goal matched in* H *are removed. As* H *contains constraints encoding all nodes and edges of* L *we successfully remove nodes and edges in* $m(L)$*. Next we add nodes and edges in* $R \setminus r(K)$*. For the CHR rule application, the constraints in* B *are added. This adds constraints encoding all nodes and edges of* R *taking into account the substitutions made for matching* H*. As* r *is an inclusion, all nodes and edges in* K *are also encoded in* $\text{encode}(R)$ *and are added to the result. We therefore keep the constraints encoding* $m(K)$ *despite temporarily removing them. We showed, that all nodes and edges present in* $\tilde{G}$ *are also contained in the modified goal store after the CHR rule application.*

It remains to be shown that also the $\text{degree}/2$ *constraints remain consistent such that the CHR rule application indeed results in* $\text{encode}(\tilde{G})$ *being added to the goal store.*

We note that for nodes $v \in L \setminus K$ *– i.e. nodes which are removed by the rule application – we only have a* degree *constraint in* H *which also gets removed. Let now* n, m *as given in Def. 5 for a* $v \in K$*. There is a corresponding constraint* $\text{degree}(T, D)$ *in* H*. For the node* v *the application of the graph production rule replaces* n *edges by* m *edges. Edges which are adjacent to* $m(v)$*, but are not in* $m(L)$ *remain unchanged. Therefore, the graph production rule changes the degree*

of $m(v)$ by $m-n$. The constraint $\text{degree}(T, D+m-n)$ *in B directly encodes this change, thus giving us a consistent degree encoding for nodes $v \in K$. Finally for nodes in $v \in R \setminus K$ which are newly added by the graph production rule we explicitly add a* degree *constraint to B with the correct degree according to Def. 5.*

"$\Leftarrow$": According to the proof of Lemma 1, the CHR match involved in the application of $\text{code}(p)$ *implies a match morphism m. We can therefore argue analogously to the above by decomposing the effects of the CHR rule application into constraints which are removed and added. By combining this with the match morphism m, we can similarly show that the correct nodes and edges are removed and added when applied to the graph transformations and that the* degree *constraints remain consistent.*

□

5 Confluence

Both graph transformation systems and constraint handling rules provide the notion of a confluence property. This property guarantees that any derivation made for an initial state results in the same final state no matter which applicable rules are applied. This section introduces the necessary definitions used for GTS and CHR confluence before comparing the two notions. It is shown how confluence checking in CHR can be adjusted to check the confluence property of a GTS encoded in CHR.

5.1 Preliminaries

Definition 7 (GTS confluence). *A GTS is called* confluent *if, for all typed graph transformations $G \overset{*}{\Rightarrow} H_1$ and $G \overset{*}{\Rightarrow} H_2$, there is a typed graph X together with typed graph transformations $H_1 \overset{*}{\Rightarrow} X$ and $H_2 \overset{*}{\Rightarrow} X$.* Local confluence *means that this property holds for all pairs of direct typed graph transformations $G \Rightarrow H_1$ and $G \Rightarrow H_2$. [2]*

A general result for rewriting systems is that it is sufficient to consider local confluence for terminating graph transformation systems. To verify local confluence we particularly need to study critical pairs and their joinability, according to this definition based on [2]:

Definition 8 (critical GTS pair). *A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} K \overset{r_2,m_2}{\Longrightarrow} P_2$ of direct typed graph transformations is called a* critical GTS pair *if it is parallel dependent, and minimal in the sense that the pair (m_1, m_2) of matches $m_1 : L_1 \to K$ and $m_2 : L_2 \to K$ is jointly surjective.*

A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} K \overset{r_2,m_2}{\Longrightarrow} P_2$ of direct typed graph transformations is called parallel independent *if $m_1(L_1) \cap m_2(L_2) \subseteq m_1(K_1) \cap m_2(K_2)$, otherwise it is called* parallel dependent*.*

A critical GTS pair $P_1 \overset{r_1,m_1}{\Longleftarrow} K \overset{r_2,m_2}{\Longrightarrow} P_2$ is called joinable *if there exists a typed graph K' together with typed graph transformations $P_1 \overset{*}{\Longrightarrow} K'$ and $P_2 \overset{*}{\Longrightarrow} K'$*

A similar notion of confluence has been developed for CHR [6]:

Definition 9 (CHR confluence). *A CHR program is called* confluent *if for all states S, S_1, and S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$, then S_1 and S_2 are joinable. Two states S_1 and S_2 are called* joinable *if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are variants.*

Analogous to a GTS, the confluence property for terminating CHR programs is determined by local confluence which can be checked through critical pairs:

Definition 10 (critical CHR pair). *Let R_1 be a simplification rule and R_2 be a (not necessarily different) rule whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head and G_i be the guard of rule $R_i (i = 1, 2)$, then a* critical ancestor state *of R_1 and R_2 is*

$$\langle H_1 \wedge A_1 \wedge H_2, (A_1 \doteq A_2) \wedge G_1 \wedge G_2 \rangle,$$

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1 \doteq A_2) \wedge G_1 \wedge G_2)$.

Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2, then the tuple (S_1, S_2) is a critical CHR pair *of R_1 and R_2. A critical CHR pair (S_1, S_2) is* joinable *if S_1 and S_2 are joinable.*

5.2 Critical Pair Properties

After defining the different notions of confluence we now further investigate the difference between critical GTS pairs and critical CHR pairs for CHR programs encoding a GTS.

Lemma 3. *If $P_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} P_2$ is a critical GTS pair, then there exists a conjunction of built-in constraints C such that $\langle \text{encode}(G), \top \rangle$ is a critical ancestor state for the critical CHR pair $(\langle \text{encode}(P_1), C \rangle, \langle \text{encode}(P_2), C \rangle)$.*

Proof. Theorem 1 guarantees that $\text{code}(p_1)$ *and* $\text{code}(p_2)$ *are applicable to the critical ancestor state.*

As m_1, m_2 are jointly surjective we know that all constraints in $\text{encode}(G)$ *are required for applying* $\text{code}(p_1)$ *and* $\text{code}(p_2)$*, i.e. there are no redundant constraints. We further know that there exist one or more constraints in* $\text{encode}(G)$ *which are required for both rule applications, as otherwise $m_1(L_1) \cap m_2(L_2) = \emptyset$ which is a contradiction to the critical GTS pair being parallel dependent. Let A be the conjunction consisting of all these constraints which are required for both rule applications.*

As $\text{code}(p_1)$ *and* $\text{code}(p_2)$ *are applicable, there are corresponding constraints A_1 and A_2 in the heads of those rules which are matched to A in a rule application. Let the heads of these rules be separated into $H_1 \wedge A_1$ and $H_2 \wedge A_2$. The match morphisms imply the existence of constraints H_1', H_2' in* $\text{encode}(G)$ *for all constraints in H_1 and H_2, respectively. These constraints have to be different from A for the rules to be applicable and they are disjoint because of the*

maximality of A. Due to the minimality of G, there are no further constraints in $\text{encode}(G)$*, therefore* $\text{encode}(G) = H_1' \wedge A \wedge H_2'$*.* $\langle \text{encode}(G), \top \rangle$ *is therefore a critical ancestor state. As* $\text{code}(p_1)$ *and* $\text{code}(p_2)$ *are both applicable to it, there is the corresponding critical pair:* $(\langle \text{encode}(P_1), A \doteq A_1 \wedge A \doteq A_2 \rangle, \langle \text{encode}(P_2), A \doteq A_1 \wedge A \doteq A_2 \rangle)$

□

Corollary 1. *If the CHR program for a terminating GTS is confluent, then all critical GTS pairs are joinable.*

Proof. As every critical GTS pair has a corresponding critical CHR pair and all critical CHR pairs are joinable, we know by the completeness property that the critical GTS pairs are also joinable.

□

Due to Corollary 1, existing confluence checkers for CHR can be used to investigate confluence of a GTS encoded in CHR. Confluence of the CHR program is a sufficient condition for confluence of the corresponding GTS however, as can be seen in the example below, there are cases in which the CHR program may not be confluent although the corresponding GTS is confluent. Similarly, if we try to transfer the confluence property of a GTS to the corresponding CHR program, we cannot succeed as in general there are too many critical CHR pairs which could cause the CHR program to be non-confluent. To improve upon this situation, we therefore introduce a weaker kind of confluence:

Definition 11 (weak confluence, valid state). *A CHR program is called* weak confluent *if for all valid states S and states* S_1 *and* S_2*: If* $S \mapsto^* S_1$ *and* $S \mapsto^* S_2$*, then* S_1 *and* S_2 *are joinable.*

A state $S = \langle G', C \rangle$ *is called* valid *if there exists a graph G such that* $G' = \text{encode}(G)$*.*

Note that especially states which encode the same node of a graph with multiple node constraints or provide multiple degree constraints for the same node are invalid states. Example 7 includes a selection of such invalid states.

Using Definition 11 we can now investigate confluence for a CHR program corresponding to a confluent GTS:

Lemma 4. *If all critical GTS pairs are joinable, the corresponding CHR program is weak confluent.*

Proof. Let $S = \langle \text{encode}(G), C \rangle$ *be a valid critical ancestor state. If the critical CHR pair resulting from S corresponds to a critical GTS pair, it is also joinable due to Theorem 2.*

Let S be a valid critical ancestor state for which the resulting critical CHR pair does not correspond to a critical GTS pair. By definition of the critical ancestor state, we know that two rules $\text{code}(r_1)$ *and* $\text{code}(r_2)$ *are applicable and by Theorem 1* r_1 *and* r_2 *are applicable to G, giving us the corresponding non-critical GTS pair* $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$*. As S does not contain redundant constraints we know that* m_1 *and* m_2 *are jointly surjective. Thus this GTS pair*

has to be parallel independent as this is the only way for it to not be a critical GTS pair. If it is parallel independent however, we know that there exists P such that $P_1 \overset{r_2,m_2'}{\Longrightarrow} P \overset{r_1,m_1'}{\Longleftarrow} P_2$ and due to the previous soundness result this implies joinability for the critical CHR pair.

□

Example 7. Consider a graph production rule for removing a loop from a node and its corresponding constraint handling rule:

$$R@\,\mathrm{node}(N), \mathrm{edge}(E, N, N), \mathrm{degree}(N, D) \Leftrightarrow \mathrm{node}(N), \mathrm{degree}(N, D-2)$$

For investigating confluence one must overlap this rule with itself which yields the following seven critical CHR ancestor states:

1. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
2. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D) \wedge \mathrm{degree}(N, D'), \top \rangle$
3. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
4. $\langle \mathrm{node}(N) \wedge \mathrm{node}(N') \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
5. $\langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D) \wedge \mathrm{degree}(N, D'), \top \rangle$
6. $\langle \mathrm{node}(N) \wedge \mathrm{node}(N') \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D) \wedge \mathrm{degree}(N, D'), \top \rangle$
7. $\langle \mathrm{node}(N) \wedge \mathrm{node}(N') \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D), \top \rangle$

States 2 and 4–7 are invalid states as they have multiple encodings of the same node or multiple degree encodings of a node. State 1 is the encoding of the corresponding critical pair for the graph production rule and state 3 is not critical because the corresponding pair of graph transformations is parallel independent:
$S_3 = \langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D), \top \rangle$
$S_3 \overset{R}{\Rightarrow} S_3^1 = \langle \mathrm{node}(N) \wedge \mathrm{edge}(E', N, N) \wedge \mathrm{degree}(N, D-2), \top \rangle$
$S_3^1 \overset{R}{\Rightarrow} S_3' = \langle \mathrm{node}(N) \wedge \mathrm{degree}(N, D-4), \top \rangle$
$S_3 \overset{R}{\Rightarrow} S_3^2 = \langle \mathrm{node}(N) \wedge \mathrm{edge}(E, N, N) \wedge \mathrm{degree}(N, D-2), \top \rangle$
$S_3^2 \overset{R}{\Rightarrow} S_3'$

Theorem 3. *All critical GTS pairs are joinable if and only if the corresponding CHR program is weak confluent.*

Proof. The first direction follows directly from Lemma 4. By Lemma 3 we know that all critical GTS pairs have corresponding critical CHR pairs with valid critical ancestor states. However, if the CHR program is weak confluent all such critical CHR pairs are joinable and thus by Theorem 2 the critical GTS pairs are joinable as well.

□

6 Conclusion

We have shown that constraint handling rules (CHR) provide an elegant way for embedding graph transformation systems (GTS). The resulting rules are very

concise and directly related to the corresponding graph production rules. There is no need for further implementations besides these production rules which makes CHR a natural choice for prototyping a GTS. We have also shown that a CHR implementation of a GTS shares many important properties with the formal GTS. Particularly the notion of confluence is similar and allows for standard CHR confluence checkers to be reused for embedded graph transformation systems with only slight adjustments.

To achieve this elegant solution, we restricted the GTS match morphisms to injective ones like [4] and [5] did. In the case of the standard nondeterministic semantics for CHR, all possible injective matches can be chosen. However, most CHR implementations make it difficult for a user to interactively modify or choose a match morphism, as it is chosen implicitly by the refined semantics of the implementation.

Apart from having a viable alternative for GTS implementations, there is room for further research. This work only considers typed graphs, but could be extended to support typed attributed graphs as well. It is also worthwhile to investigate further similarities between GTS and CHR by transferring existing results from one model to the other. For example, [2] provides criteria for layered graph transformation systems under which termination can be guaranteed and which might be applicable to CHR as well.

References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (October 1998) 95–138
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag (2006)
3. Sadiq, W., Orlowska, M.E.: Applying graph reduction techniques for identifying structural conflicts in process models. In: CAiSE '99: Proceedings of the 11th International Conference on Advanced Information Systems Engineering, London, UK, Springer-Verlag (1999) 195–209
4. Bakewell, A., Plump, D., Runciman, C.: Specifying pointer structures by graph reduction. In: AGTIVE. (2003) 30–44
5. Habel, A., Plump, D.: Relabelling in graph transformation. In: Proc. Int. Conference on Graph Transformation (ICGT 2002). Lecture Notes in Computer Science, Springer-Verlag (2002)
6. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer-Verlag (2003)
7. Frühwirth, T.: As time goes by: Automatic complexity analysis of simplification rules. In: 8th International Conference on Principles of Knowledge Representation and Reasoning, Toulouse, France (2002)