*TLP* **11** (1): 65–109, 2011. © Cambridge University Press 2010
doi:10.1017/S1471068410000438 First published online 20 July 2010


# *Analysing graph transformation systems through constraint handling rules*

FRANK RAISER and THOM FRÜHWIRTH
*Faculty of Engineering and Computer Sciences, Ulm University, Germany*
(*e-mail:* {Frank.Raiser|Thom.Fruehwirth}@uni-ulm.de)

*submitted 25 September 2009; revised 18 April 2010; accepted 6 June 2010*

**Abstract**

Graph transformation systems (GTS) and constraint handling rules (CHR) are non-deterministic rule-based state transition systems. CHR is well known for its powerful confluence and program equivalence analyses, for which we provide the basis in this work to apply them to GTS. We give a sound and complete embedding of GTS in CHR, investigate confluence of an embedded GTS and provide a program equivalence analysis for GTS via the embedding. The results confirm the suitability of CHR-based program analyses for other formalisms embedded in CHR.

*KEYWORDS*: Graph transformation systems, Constraint handling rules, Program analysis

## 1 Introduction

Graph transformation systems (GTS) 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 (Blostein *et al*. 1995; Ehrig *et al*. 2006). GTS see widespread use in many applications (Ehrig *et al*. 2006), and hence performing program analysis on them is becoming more important.

Constraint handling rules (CHR) (Frühwirth 2009) on the other side allows for rapid prototyping of constraint-based algorithms. Besides constraint reasoning, CHR has been used for such diverse applications as type system design for Haskell (Sulzmann *et al*. 2006), time tabling (Abdennadher and Marte 2000), computational linguistics (Dahl and Maharshak 2009), chip card verification (Pretschner *et al*. 2004), computational biology (Bavarian and Dahl 2006) and decision support for cancer diagnosis (Barranco-Mendoza 2005). Essentially, CHR performs guarded multiset rewriting, extended by a complete and decidable constraint theory. A specific strength of CHR is the wide array of available program analyses. Other formalisms have been embedded in CHR in order to compare and mutually benefit from different analysis approaches (cf. Section 6). In this work, we extend this line of research by embedding GTS in CHR and comparing confluence and operational equivalence analysis methods.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

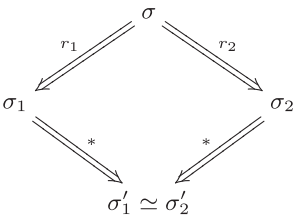



Fig. 1. Confluence property for rules $r_1$ and $r_2$.

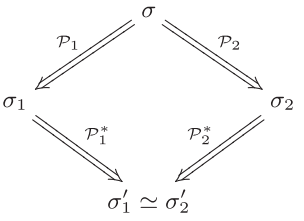



Fig. 2. Operational equivalence for programs $\mathscr{P}_1$ and $\mathscr{P}_2$.

First, we embed GTS in CHR (Raiser 2007) in Section 3. This encoding is intuitive and offers a clear one-to-one correspondence between GTS and CHR rules. Our proposed encoding characterizes a subset of CHR that closely corresponds to GTS, and furthermore we prove its soundness and completeness. Then, we show that CHR is capable of expressing infinite numbers of graphs, which we will call *partial graphs*, and their transformations in a finite way, thus facilitating program analysis.

In non-deterministic rule-based systems, like GTS and CHR, two or more rules can be applied to a state $\sigma$. An interesting property in that respect is the notion of *confluence*, which holds, if for any case in which two rules are applicable there exist computations yielding the same, or equivalent, results. This situation is displayed in Figure 1, which due to its shape is referred to as the *diamond property*.

For terminating CHR programs a decidable automatic confluence test exists, based on research in the area of term-rewriting (Baader and Nipkow 1998). However as shown in Plump (2005), an analogous approach fails for GTS. Therefore, confluence analysis is an important example for a program analysis of a GTS with methods from CHR. In Section 4 we show that the confluence test for CHR coincides with the strongest known sufficient criterion for confluence of a GTS (Raiser and Frühwirth 2009b).

In Section 5 we examine operational equivalence (Abdennadher and Frühwirth 1999) as a second example of a program analysis that is available for CHR and can be applied to GTS. Operational equivalence, intuitively, decides if two programs can compute equivalent results when given the same input, as shown in Figure 2. The diamond shape in Figure 2 emphasizes the similarity to confluence, which is also found in the respective program analysis methods. We introduce operational equivalence in the GTS context in analogy to CHR (Raiser and Frühwirth 2009a). Then, we prove that deciding operational equivalence of a CHR program, derived from a GTS, is a sufficient criterion for operational equivalence of the corresponding GTS.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

An interesting application of this result is the possibility to detect and remove redundant rules using the test for operational equivalence. Redundant rules of GTS have been formally defined in Kreowski and Valiente (2000); however, to the best of our knowledge, this is the first available algorithm for detecting them in a GTS.

This work presents a unified treatment and considerable extension of previously published works (Raiser 2007, 2009; Raiser *et al.* 2009; Raiser and Frühwirth 2009a, 2009b). In Raiser *et al.* (2009) a formal treatment of CHR state equivalence is provided and, derived from that, a simplified formulation of the operational semantics of CHR. This novel formulation allows us to unify our previous works while simplifying presentation and formal proofs significantly. Furthermore, the state equivalence definition from Raiser *et al.* (2009) is the basis for new insights on CHR states that encode graphs.

## 2 Preliminaries

In this section, we introduce the required formalisms for GTS in Section 2.1 and CHR in Section 2.2.

### *2.1 Graph transformation system*

The following definitions for graphs and GTS have been adapted from Ehrig *et al.* (2006).

*Definition 2.1* (*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 functions $\mathrm{src}, \mathrm{tgt} : E \to V$ specifying source and target of an edge, respectively. A *type graph* $TG$ is a graph with unique labels for all nodes and edges.

For simplicity, we avoid an additional label function in favour of identifying variable names with 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 $\mathrm{src}, \mathrm{tgt}$ functions. We further define the degree of a node as $\deg : V \to \mathbb{N}, v \mapsto \#\{e \in E \mid \mathrm{src}(e) = v\} + \#\{e \in E \mid \mathrm{tgt}(e) = v\}$. As there may be multiple graphs containing the same node, we use $\deg_G(v)$ to specify the degree of a node $v$ with respect to the graph $G$. When the context graph is clear the subscript is omitted.

In this work, we consider typed graphs, i.e. graphs in which nodes and edges are assigned types from a type graph.

*Definition 2.2* (*graph morphism, typed graph*)
Given graphs $G_1, G_2$ with $G_i = (V_i, E_i, \mathrm{src}_i, \mathrm{tgt}_i)$ for $i = 1, 2$ a *graph morphism* $f : G_1 \to G_2, f = (f_V, f_E)$ consists of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve the source target functions, i.e. $f_V \circ \mathrm{src}_1 = \mathrm{src}_2 \circ f_E$ and $f_V \circ \mathrm{tgt}_1 = \mathrm{tgt}_2 \circ f_E$.

A graph morphism $f$ is *injective* (or *surjective*) if both functions $f_V, f_E$ are injective (or surjective, respectively); $f$ is called *isomorphic* if it is bijective. $f$ is called an *inclusion* if $f_V(V_1) \subseteq V_1$ and $f_E(E_1) \subseteq E_1$. When the context is clear, we simply refer to graph morphisms as morphisms.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

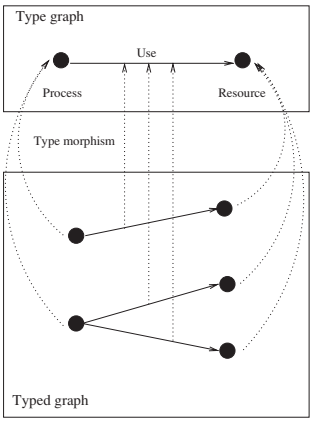



Fig. 3. Example of a type graph and typed graph.

A *typed graph* $G$ is a tuple $(V, E, \text{src}, \text{tgt}, \text{type}, TG)$, where $(V, E, \text{src}, \text{tgt})$ is a graph, $TG$ is a type graph and type is a graph morphism with $\text{type} = (\text{type}_V, \text{type}_E)$ and $\text{type}_V : V \to TG_V, \text{type}_E : E \to TG_E$.

For a typed graph $G = (V, E, \text{src}, \text{tgt}, \text{type}, TG)$ we define a *subgraph* $H$ as a typed graph $(V', E', \text{src}', \text{tgt}', \text{type}', TG)$ such that $V' \subseteq V \wedge E' \subseteq E \wedge \text{src}' = \text{src}\,|_{E'} \wedge \text{tgt}' = \text{tgt}\,|_{E'} \wedge \text{type}'_V = \text{type}_V\,|_{V'} \wedge \text{type}'_E = \text{type}_E\,|_{E'}$ with $\forall e \in E'.\, \text{src}'(e) \in V' \wedge \text{tgt}'(e) \in V'$.

*Example 2.1*

Figure 3 shows an example for a type graph and a corresponding typed graph. The type graph at the top defines two types of nodes: processes and resources. Furthermore, it defines *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 graph morphism is represented by the dotted lines, showing how the nodes are typed as processes or resources, respectively.

*Definition 2.3* (*GTS, rule*)

A *graph transformation system* is a tuple consisting of a type graph and a set of graph production rules. A *graph production rule* – simply called *rule* if the context is clear – is a tuple $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ of graphs $L, K$ and $R$ 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 are applied. This work is based on the double-pushout approach (DPO) as defined in Ehrig *et al.* (2006). Most notably, we require a *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 $G \overset{p,m}{\Longrightarrow} H$. $H$ is given mathematically by constructing $D$ as shown in Figure 4, such that (1) and (2) are pushouts in the category of typed graphs. Intuitively, the graph $L$ is matched to a subgraph of $G$ and its occurrence in $G$ is then replaced by the graph $R$. The intermediate graph $K$ is the *context graph*, which contains the nodes and

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

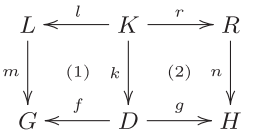



Fig. 4. Double-pushout approach.

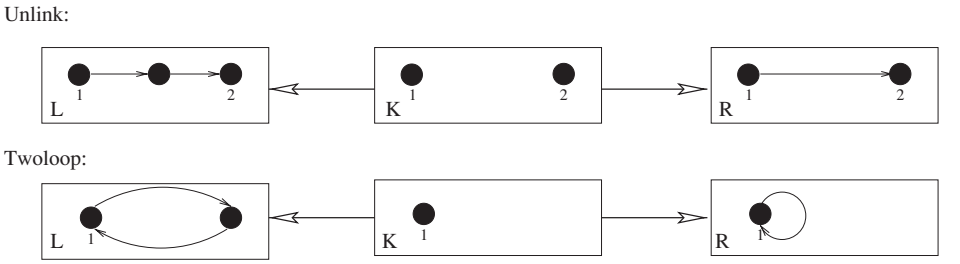



Fig. 5. Graph transformation system for recognizing cyclic lists.

edges in both $L$ and $R$, i.e. all nodes and edges matched to $K$ remain during the transformation.

A graph production rule $p$ can only be applied to a host graph $G$ if the following gluing condition is satisfied. In fact, Ehrig *et al.* (2006) shows that $D$ and the pushout (1) exist if and only if this gluing condition is satisfied. It is based on the following three sets (Ehrig *et al*. 2006):

- gluing points: $GP = l(K)$
- identification points: $IP = \{v \in V_L \mid \exists w \in V_L, w \neq v : m(v) = m(w)\} \cup \{e \in E_L \mid \exists f \in E_L, e \neq f : m(e) = m(f)\}$
- dangling points: $DP = \{v \in V_L \mid \exists e \in E_G \setminus m(E_L) : \mathrm{src}_G(e) = m(v) \vee \mathrm{tgt}_G(e) = m(v)\}$.

*Definition 2.4* (*gluing condition*)
The *gluing condition* is defined as $IP \cup DP \subseteq GP$.

If the gluing condition is satisfied for a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ the application of the rule consists of transforming $G$ into $H$ by performing the construction described above. An 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 2.2*
Figure 5 shows two graph production rules in a shorthand notation that defines the morphisms $l$ and $r$ implicitly by the labels of the nodes which are mapped onto each other. The resulting graph transformation system is implicitly defined over the simple type graph consisting only of a single node with a loop, depicted in Figure 6.

The two rules constitute a graph transformation system for detecting cyclic lists. The basic idea of the *unlink* rule is to remove intermediate nodes of the list, while the *twoloop* rule replaces the cyclic list consisting of two nodes by a single node with

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press



Fig. 6. Simple type graph consisting of a node and edge.

a loop. Note that application of the *twoloop* rule requires that no additional edges are adjacent to the removed node. Such *dangling edges* are discussed in more detail in Section 3.

To detect if a host graph is a cyclic list, the GTS is applied to the host graph until exhaustion, i.e. until no rule is applicable anymore. The initial host graph then is a cyclic list if and only if the final graph consists of a single node with a loop (cf. Bakewell *et al.* 2003).

In general, the match morphism $m$ can be non-injective. However, for the remainder of this work we only consider injective match morphisms, which have the advantage that the set $IP$ of identification points is guaranteed to be $\emptyset$. Furthermore, non-injective match morphisms can be simulated as follows: given a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a non-injective match morphism $m$ it holds $\forall v, w \in V_L, v \neq w$ with $m(v) = m(w)$ that the rule is only applicable, if $v, w \in l(V_K)$, i.e. only nodes which are not removed by the rule application are allowed to be matched non-injectively – otherwise $IP \nsubseteq GP$. Therefore, it is possible to add another rule $p'$ which is derived from $p$ by merging the nodes $v$ and $w$ into a node $v_w$ in all three graphs of the rule. Thus, the non-injective matching with $m(v) = m(w)$ can be simulated by injectively matching $v_w$ to $m(v_w)$, where $m(v_w)$ is the same node in $G$ as $m(v)$. The same argumentation holds for edges, analogously. Therefore, we can restrict ourselves to injective match morphisms by extending the set of rules with new rules for all possible merges of nodes and edges in the graph $K$. This simplifies the generic gluing condition to $DP \subseteq GP$.

Finally, we require the following definition of the track morphism (Plump 1995). Intuitively, the track morphism is defined for a node or edge, if it is not removed by the rule application.

*Definition 2.5* (*track morphism*)
Given $G \Rightarrow H$ the *track morphism* $\mathrm{tr}_{G \Rightarrow H} : G \rightarrow H$ is the partial graph morphism defined by

$$\mathrm{tr}_{G \Rightarrow H}(x) = \begin{cases} g(f^{-1}(x)) & \text{if } x \in f(D), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here $f : D \rightarrow G$ and $g : D \rightarrow H$ are the morphisms in the lower row of the pushout (1) in Figure 4 and $f^{-1} : f(D) \rightarrow D$ maps each item $f(x)$ to $x$.

The track morphism of a derivation $\Delta : G_0 \Rightarrow^* G_n$ is defined by $\mathrm{tr}_\Delta = \mathrm{id}_{G_0}$ if $n = 0$ and $\mathrm{tr}_\Delta = \mathrm{tr}_{G_1 \Rightarrow^* G_n} \circ \mathrm{tr}_{G_0 \Rightarrow G_1}$ otherwise, where $\mathrm{id}_{G_0}$ is the identity morphism on $G_0$.

### 2.2 Constraint handling rules

This section presents the syntax and operational semantics of CHR (Frühwirth 2009; Sneyers *et al.* 2010). Constraints are first-order predicates which we separate into

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

*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. The notation $c/n$, where $c$ is called the *constraint symbol* and $n$ the *arity*, is used for both types of constraints.

Its semantics is based on an underlying complete *constraint theory* $\mathscr{CT}$ on built-in constraints for which satisfiability and entailment are decidable. In general, CHR allows arbitrary constraint theories for $\mathscr{CT}$, requiring only that it contains at least Clark's equality theory for syntactic equality. In addition to that, in this work we also require $\mathscr{CT}$ to cover the elementary arithmetic operations $+$ and $-$. Furthermore, $\top$ denotes the built-in which is always true and $\bot$ denotes false, respectively.

The survey (Sneyers *et al*. 2010) provides an overview over the different techniques used in CHR implementations and the book (Frühwirth 2009) details the different available operational semantics for CHR. In this work we abstract from specific implementations and rely on the operational semantics given in Raiser *et al.* (2009), which corresponds to the *very abstract operational semantics* in Frühwirth (2009).

CHR is a state transition system over the set of states given in the following definition.

*Definition 2.6* (*CHR states*)
A (*CHR*) *state*) is a tuple

$$\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle.$$

$\mathbb{G}$ is a multiset of user-defined constraints called the *goal* (or (*user-defined*) *constraint store*), $\mathbb{B}$ is a conjunction of built-in constraints called the *built-in* (*constraint*) *store* and $\mathbb{V}$ is the set of *global variables*.

In this work $\sigma, \tau, \ldots$ denote CHR states and $\Sigma$ denotes the set of all CHR states.

The following definition introduces the different types of variables we distinguish for a given CHR state.

*Definition 2.7* (*Variable Types*)
For the variables occurring in a state $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ we distinguish three different types:

(1) a variable $v \in \mathbb{V}$ is called a *global* variable,
(2) a variable $v \notin \mathbb{V}$ is called a *local* variable,
(3) a variable $v \notin (\mathbb{V} \cup \mathrm{vars}(\mathbb{G}))$ is called a *strictly local* variable.

The following equivalence relation $\equiv$ between CHR states (Raiser *et al*. 2009) is an important tool that facilitates a succinct definition of the operational semantics of CHR and simplifies proofs.

*Definition 2.8* (*State Equivalence*)
Equivalence between CHR states is the smallest equivalence relation $\equiv$ over CHR states that satisfies the following conditions:

(1) (*Substitution*)

$$\langle \mathbb{G}, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}\,[x/t]\,, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle.$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

(2) (*Transformation of the Constraint Store*) If $\mathscr{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$, where $\bar{s}, \bar{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$, respectively, then:

$$\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}, \mathbb{B}', \mathbb{V} \rangle.$$

(3) (*Omission of Non-Occurring Global Variables*) If $X$ is a variable that does not occur in $\mathbb{G}$ or $\mathbb{B}$ then:

$$\langle \mathbb{G}, \mathbb{B}, \{X\} \cup \mathbb{V} \rangle \equiv \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle.$$

(4) (*Equivalence of Failed States*)

$$\langle \mathbb{G}, \bot, \mathbb{V} \rangle \equiv \langle \mathbb{G}', \bot, \mathbb{V} \rangle.$$

The following lemma presents basic properties of this equivalence relation:

*Lemma 1* (*Properties of State Equivalence* (*Raiser* et al. *2009*))
The equivalence relation over CHR states, given in Definition 2.8, has the following properties:

(1) (*Renaming of Local Variables*) Let $x, y$ be variables such that $x, y \notin \mathbb{V}$ and $y$ does not occur in $\mathbb{G}$ or $\mathbb{B}$:

$$\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}\left[x/y\right], \mathbb{B}\left[x/y\right], \mathbb{V} \rangle.$$

(2) (*Partial Substitution*) Let $\mathbb{G}\,[x \wr t]$ be a multiset, where *some* occurrences of $x$ are substituted with $t$:

$$\langle \mathbb{G}, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}\,[x \wr t], x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle.$$

(3) (*Logical Equivalence*) If

$$\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}', \mathbb{B}', \mathbb{V}' \rangle,$$

then $\mathscr{CT} \models \exists \bar{y}.\mathbb{G} \wedge \mathbb{B} \leftrightarrow \exists \bar{y}'.\mathbb{G}' \wedge \mathbb{B}'$, where $\bar{y}, \bar{y}'$ are the local variables of $\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \langle \mathbb{G}', \mathbb{B}', \mathbb{V}' \rangle$, respectively.

Decidability of state equivalence is a result of the following theorem from Raiser *et al.* (2009):

*Theorem 2* (*Criterion for* $\equiv$ (*Raiser* et al. *2009*))
Let $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \sigma' = \langle \mathbb{G}', \mathbb{B}', \mathbb{V} \rangle$ be CHR states with local variables $\bar{y}, \bar{y}'$ that have been renamed apart.

$$\sigma \equiv \sigma' \text{ iff } \mathscr{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B})).$$

As CHR is a rule-based programming language we now introduce the different types of possible CHR rules.

*Definition 2.9* (*CHR Rules, CHR Program*)
For multisets $H_1, H_2, B_c$ of user-defined constraints with $H_1, H_2 \neq \emptyset$ and conjunctions $G, B_b$ of built-in constraints a CHR *simpagation* rule is of the form

$$H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b.$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

For the case $H_1 = \emptyset$ we call the rule a *simplification* rule and denote it as

$$H_2 \Leftrightarrow G \mid B_c, B_b,$$

and for the case $H_2 = \emptyset$ we call the rule a *propagation* rule and denote it as

$$H_1 \Rightarrow G \mid B_c, B_b.$$

If $G = \top$ it can be omitted together with the ' $\mid$'.

A *CHR program* is a set of CHR rules.

Next, we define the operational semantics of CHR by introducing its transition relation $\rightarrowtail$ based on the formulation given in Raiser *et al.* (2009), which relies on equivalence classes of CHR states. In the remainder of this work we take the liberty of notationally identifying a CHR state $\sigma$ with its equivalence class $[\sigma]$. Furthermore, we simplify multiset expressions like $\{a\} \uplus \{b\}$ to $a \uplus b$ or $a, b$.

*Definition 2.10* (*Operational Semantics*)
For a CHR program $\mathscr{P}$ we define the state transition system $(\Sigma/\equiv, \rightarrowtail)$ as follows. The application of a rule $r \in \mathscr{P}$ assumes a copy of it that contains only fresh variables.

$$\frac{r \ @ \ H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b}{[\langle H_1 \uplus H_2 \uplus \mathbb{G}, G \wedge \mathbb{B}, \mathbb{V}\rangle] \rightarrowtail [\langle H_1 \uplus B_c \uplus \mathbb{G}, G \wedge B_b \wedge \mathbb{B}, \mathbb{V}\rangle]}$$

Simplification rules are only syntactically different, but operate as described by Definition 2.10 with $H_1 = \emptyset$, respectively. Note that propagation rules lead to trivial non-termination in this formulation, however that is no problem here, because the work at hand requires no propagation rules.

A rule $r \in \mathscr{P}$ is *applicable* to a state $\sigma$ if and only if there exists a state $\tau$ such that $\sigma \rightarrowtail \tau$. We say that a state $\sigma$ is *final* if and only if there exists no state $\tau$ with $\sigma \rightarrowtail \tau$. As usual, $\rightarrowtail^*$ denotes the reflexive-transitive closure of $\rightarrowtail$. When we want to emphasize that a transition uses a specific rule $r$ we denote this by $\rightarrowtail^r$. When discussing multiple programs, $\rightarrowtail_{\mathscr{P}}$ denotes a transition using a rule of program $\mathscr{P}$.

*Example 2.3* (*Example Computation*)
In this comprehensive example, we present a complete computation in CHR. Readers already familiar with CHR may want to skip this.

The following rule (Frühwirth 2009) is a program for computing the minimum of a multiset of numbers:

$$\min(N) \backslash \min(M) \Leftrightarrow N \leqslant M \mid \top.$$

Intuitively, two min constraints are matched and the one with the larger number is removed. We will now walk through the detailed computation of running the following input $\sigma$ on the above program, in order to determine the minimum of the numbers 1, 3 and 4:

$$\sigma = \langle \min(1) \uplus \min(3) \uplus \min(X), X = 4, \{X\}\rangle.$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

First, we take a fresh copy of the rule as demanded by Definition 2.10:

$$\min(N_1)\backslash \min(M_1) \Leftrightarrow N_1 \leqslant M_1 \mid \top.$$

Next, we apply Definition 2.8 in order to show that $\sigma$ is contained in the equivalence class required for applying this rule (we use $\mathbb{V} = \{X\}$ here):

$$\begin{aligned}
\sigma &\overset{\mathscr{CT}}{\equiv} \langle \min(1) \uplus \min(3) \uplus \min(X), N_1 \leqslant M_1 \wedge X = 4 \wedge N_1 = 1 \wedge M_1 = 3, \mathbb{V} \rangle \\
&\overset{\text{Subst}}{\equiv} \langle \min(N_1) \uplus \min(M_1) \uplus \min(X), N_1 \leqslant M_1 \wedge X = 4 \wedge N_1 = 1 \wedge M_1 = 3, \mathbb{V} \rangle \\
&= \langle \min(N_1) \uplus \min(M_1) \uplus \mathbb{G}, N_1 \leqslant M_1 \wedge \mathbb{B}, \mathbb{V} \rangle.
\end{aligned}$$

Hence, all conditions for Definition 2.10 are satisfied, so we can apply the rule to the equivalence class of $\sigma$, getting $\sigma \rightarrowtail \tau$, or more precisely, $[\sigma] \rightarrowtail [\tau]$:

$$\begin{aligned}
\sigma &\rightarrowtail \langle \min(N_1) \uplus \mathbb{G}, N_1 \leqslant M_1 \wedge \top \wedge \mathbb{B}, \mathbb{V} \rangle \\
&= \langle \min(N_1) \uplus \min(X), N_1 \leqslant M_1 \wedge \top \wedge X = 4 \wedge N_1 = 1 \wedge M_1 = 3, \mathbb{V} \rangle \\
&\overset{\text{Subst}}{\equiv} \langle \min(1) \uplus \min(X), N_1 \leqslant M_1 \wedge \top \wedge X = 4 \wedge N_1 = 1 \wedge M_1 = 3, \mathbb{V} \rangle \\
&\overset{\mathscr{CT}}{\equiv} \langle \min(1) \uplus \min(X), X = 4, \mathbb{V} \rangle = \tau.
\end{aligned}$$

Next, we repeat this procedure for another application of the above rule, based on the following fresh copy:

$$\min(N_2)\backslash \min(M_2) \Leftrightarrow N_2 \leqslant M_2 \mid \top.$$

This results in the expected answer that 1 is the minimum of the numbers 1, 3 and 4:

$$\begin{aligned}
\tau &\overset{\mathscr{CT}}{\equiv} \langle \min(1) \uplus \min(X), N_2 \leqslant M_2 \wedge N_2 = 1 \wedge M_2 = X \wedge X = 4, \mathbb{V} \rangle \\
&\overset{\text{Subst}}{\equiv} \langle \min(N_2) \uplus \min(M_2), N_2 \leqslant M_2 \wedge N_2 = 1 \wedge M_2 = X \wedge X = 4, \mathbb{V} \rangle \\
&\rightarrowtail \langle \min(N_2), N_2 \leqslant M_2 \wedge \top \wedge N_2 = 1 \wedge M_2 = X \wedge X = 4, \mathbb{V} \rangle \\
&\overset{\text{Subst}}{\equiv} \langle \min(1), N_2 \leqslant M_2 \wedge \top \wedge N_2 = 1 \wedge M_2 = X \wedge X = 4, \mathbb{V} \rangle \\
&\overset{\mathscr{CT}}{\equiv} \langle \min(1), X = 4, \mathbb{V} \rangle.
\end{aligned}$$

We can also witness the difference between global and local variables in this computation. While the variable $X$ is no longer used in a CHR constraint in the final state, we still have to keep track of the information $X = 4$, because it is a global variable. The auxiliary variables $N_1, M_1, \ldots$ instead, are local when used in a CHR constraint and strictly local, when only occurring in the built-in store. In the latter case we may replace the built-in store by a logically equivalent representation that removes the strictly local variables.

## 3 Embedding GTS in CHR

In this section we encode rules of a graph transformation system as CHR rules and discuss how host graphs are encoded in CHR to work with these rules. Section 3.1 defines the necessary encoding and presents an example computation in CHR. We then analyse formal properties of GTS embedded in CHR in Section 3.2. Finally, Section 3.3 discusses the suitability of this encoding for program analysis and variations of the encoding.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

In this work, we assume that the CHR programs resulting from encoding a GTS are executed only with encodings of graphs. Naturally, we may provide the CHR programs with completely different inputs or inconsistently encoded graphs. It is clear that we cannot expect any meaningful results from such computations, hence, for the remainder of this work we restrict all observations to programs and states that correspond to GTS and graphs. We formalize this restriction in Section 3.2 by means of an invariant. Therefore, on one hand any state that violates the invariant will not be considered as input, and, on the other hand, any graph can be encoded into a state that satisfies the invariant. We show in Section 3.2.2 that execution of the encoded GTS in CHR for invariant-satisying states always leads to results that also satisfy the invariant. In other words, when providing a graph as input to the CHR program, the result will also be a graph, as is to be expected.

### *3.1 CHR encoding of a GTS*

First, we determine the necessary constraint symbols for encoding rule and host graphs. At this point we require the GTS to be typed, so this can be directly inferred from the corresponding type graph as explained in Definition 3.1. Note that this is not a restriction though, as every untyped graph can be typed over the type graph consisting of a single node with a loop (cf. Figure 6).

*Definition 3.1* (*type graph encoding*)
For a type graph $TG$ we define the set $\mathscr{C}$ of required constraint symbols to encode graphs typed over $TG$ as the minimal set satisfying:

- If $v \in V_{TG}$ then $v/2 \in \mathscr{C}$.
- If $e \in E_{TG}$ then $e/3 \in \mathscr{C}$.

We assume that all constraints introduced by Definition 3.1 have unique names. Furthermore, for graphs to be encoded with these constraints, we associate elements of the set $V$ of nodes with integer numbers or letters that can be used as arguments.

*Definition 3.2* (*typed graph encoding*)
We define the following helpful mappings for an infinite set of variables VARS:

- $\text{type}_G(x)$ denotes the corresponding constraint symbol for encoding a node or edge of the given type.
- $\text{var} : G \to \text{VARS}, x \mapsto X_x$ such that $X_x$ is a unique variable associated to $x$, i.e. var is injective for $X$ being the set of all graph nodes and edges.
- $\text{dvar} : G \to \text{VARS}, x \mapsto X_x$ such that $X_x$ is a unique variable associated to $x$, i.e. dvar is injective for $X$ being the set of all graph nodes and edges and different from var.

Using these mappings we define the following encoding of graphs:

$$\texttt{chr}_G(E, x) = \begin{cases} \text{type}_G(x)(\text{var}(x), \text{deg}_G(x)) & \text{if } x \in V_G \wedge E = \text{ground}, \\ \text{type}_G(x)(\text{var}(x), \text{dvar}(x)) & \text{if } x \in V_G \wedge E = \text{keep}, \\ \text{type}_G(x)(\text{var}(x), \text{var}(\text{src}(x)), \\ \quad \text{var}(\text{tgt}(x))) & \text{if } x \in E_G. \end{cases}$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

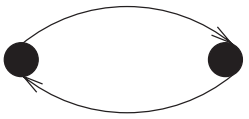


Fig. 7. Cyclic graph consisting of two nodes.

We use the notations $\mathtt{chr}(\text{ground}, G) = \{\mathtt{chr}_G(\text{ground}, x) \mid x \in G\}$ as well as $\mathtt{chr}(\text{keep}, G) = \{\mathtt{chr}_G(\text{keep}, x) \mid x \in G\}$. Furthermore, we omit the index $G$ if the context is clear. We call $\text{dvar}(v)$ the *degree variable* for a node $v$.

A host graph $G$ is encoded in CHR as $\langle \mathtt{chr}(\text{ground}, G), \top, \mathbb{V}\rangle$, where $\mathbb{V}$ can be chosen freely.

*Example 3.1* (*cont*)
For our example of the GTS for recognizing cyclic lists we assume the type graph in Figure 6. Based on this type graph we need the constraints node /2 and edge /3. The host graph $G$ given in Figure 7 that contains a cyclic list consisting of exactly two nodes is encoded in $\mathtt{chr}(\text{ground}, G)$ as

$$\text{node}(N_1, 2), \text{node}(N_2, 2), \text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_1).$$

The same graph $G$ encoded in $\mathtt{chr}(\text{keep}, G)$ has the following form:

$$\text{node}(N_1, D_1), \text{node}(N_2, D_2), \text{edge}(E_1, N_1, N_2), \text{edge}(E_2, N_2, N_1).$$

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

*Definition 3.3* (*GTS rule in CHR*)
For a graph production rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ from a GTS we define $\varrho(p) = (p \mathbin{@} C_L \Leftrightarrow C_R^u, C_R^b)$ with

- $C_L = \{\mathtt{chr}_L(\text{keep}, x) \mid x \in K\} \uplus \{\mathtt{chr}_L(\text{ground}, x) \mid x \in L \setminus K\}$
- $C_R^u = \{\mathtt{chr}_R(\text{ground}, x) \mid x \in R \setminus K\} \uplus \{\mathtt{chr}_R(\text{keep}, e) \mid e \in E_K\}$
  $\uplus\{\mathtt{chr}_R(\text{keep}, v') \mid v \in V_K\}$
- $C_R^b = \{\text{var}(v) = \text{var}(v') \wedge \text{dvar}(v') = \text{dvar}(v) - \deg_L(v) + \deg_R(v) \mid v \in V_K\}$.

A CHR program that is created from a GTS according to the above definition, will be referred to as a *GTS–CHR program* for the remainder of this work.

*Example 3.2* (*cont.*)
As an example, consider the second rule from Example 2.2, which reduces two cyclic nodes to a single node with a loop. Its encoding as a CHR simplification rule is given below:

$$\begin{aligned} \text{twoloop} \mathbin{@}\ & \text{node}(N_1, D_1) \uplus \text{node}(N_2, 2) \uplus \\ & \text{edge}(E_1, N_1, N_2) \uplus \text{edge}(E_2, N_2, N_1) \\ & \Leftrightarrow \\ & \text{node}(N_1', D_1') \uplus \text{edge}(E_3, N_1, N_1), N_1' = N_1 \wedge D_1' = D_1 - 2 + 2. \end{aligned}$$

Note that it is also possible to simplify this encoding, as explained later in Section 3.3.2.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

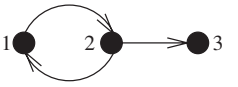



Fig. 8. Graph with a dangling edge if node 2 is removed by the *twoloop* rule.

When applying a GTS rule the gluing condition has to be satisfied. Due to our restriction to injective match morphisms, the gluing condition is violated if there exists $x \in DP$ with $x \notin GP$. Intuitively, when a node gets deleted by a rule, the corresponding node in the host graph may have an edge adjacent to it which is 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 3.4* (*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 $m(v) = \mathrm{src}(e) \vee m(v) = \mathrm{tgt}(e)$.

*Example 3.3* (*cont.*)
Consider the *twoloop* rule given in Example 3.2, along with the following encoded host graph shown in Figure 8:

$$\mathrm{node}(V_1, 2), \mathrm{node}(V_2, 3), \mathrm{node}(V_3, 1),$$
$$\mathrm{edge}(E_1, V_1, V_2), \mathit{edge}(E_2, V_2, V_1), \mathrm{edge}(E_3, V_2, V_3).$$

Applying the *twoloop* rule to this graph to remove the node $V_2$ would leave the edge $E_3$ dangling. However, this is avoided as the encoding of the *twoloop* rule contains the following constraint in its head: $\mathrm{node}(N_2, 2)$. Hence, only a node with a degree of exactly 2 can be removed by this rule. Nevertheless, the rule can be applied with $N_2 = V_1$ as the node $V_1$ has the required degree of 2.

### 3.1.1 Example computation

In this section we provide a complete computation for our cyclic list example to demonstrate how our encoding works. The following two rules are the CHR encoding of the rules in Figure 5:

$$\begin{aligned} \mathit{unlink}\ @\quad & \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N, 2) \uplus \mathrm{node}(N_2, D_2) \uplus \\ & \mathrm{edge}(E_1, N_1, N) \uplus \mathrm{edge}(E_2, N, N_2) \\ & \Leftrightarrow \\ & \mathrm{node}(N_1', D_1') \uplus \mathrm{node}(N_2', D_2') \uplus \mathrm{edge}(E, N_1, N_2), \\ & N_1' = N_1 \wedge N_2' = N_2 \wedge D_1' = D_1{+}1{-}1 \wedge D_2' = D_2{+}1{-}1 \end{aligned}$$

$$\begin{aligned} \mathit{twoloop}\ @\quad & \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N, 2) \uplus \\ & \mathrm{edge}(E_1, N_1, N) \uplus \mathrm{edge}(E_2, N, N_1) \\ & \Leftrightarrow \\ & \mathrm{node}(N_1', D_1') \uplus \mathrm{edge}(E, N_1, N_1), \\ & N_1' = N_1 \wedge D_1' = D_1{+}2{-}2. \end{aligned}$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

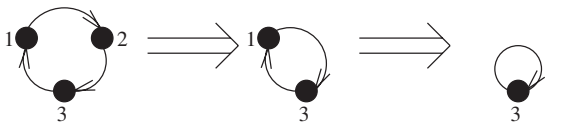



Fig. 9. Example computation.

The following state $\sigma$ encodes a cycle consisting of three nodes. The following computation is depicted in Figure 9. To demonstrate computations on partially defined graphs, further discussed in Section 3.3, the degree of the third node is left uninstantiated:

$$\begin{aligned}\sigma = \langle & \mathrm{node}(N_1, 2) \uplus \mathrm{node}(N_2, 2) \uplus \mathrm{node}(N_3, D_3) \uplus \\ & \mathrm{edge}(E_1, N_1, N_2) \uplus \mathrm{edge}(E_2, N_2, N_3) \uplus \mathrm{edge}(E_3, N_3, N_1), \\ & \top, \{N_1, N_2, N_3, E_1, E_2, E_3, D_3\}\rangle.\end{aligned}$$

Rule *unlink* is applied to state $\sigma$ resulting in the state

$\langle \mathrm{node}(N_1', D_1') \uplus \mathrm{node}(N_3', D_3') \uplus \mathrm{edge}(E, N_1, N_3) \uplus \mathrm{edge}(E_3, N_3, N_1),$
$N_1' = N_1 \wedge D_1' = 2{+}1{-}1 \wedge N_3' = N_3 \wedge D_3' = D_3{+}1{-}1, \{N_1, N_2, N_3, E_1, E_2, E_3, D_3\}\rangle,$

which is equivalent to state $\sigma'$:

$\sigma' = \langle \mathrm{node}(N_1, 2) \uplus \mathrm{node}(N_3, D_3) \uplus \mathrm{edge}(E, N_1, N_3) \uplus \mathrm{edge}(E_3, N_3, N_1),$
$\top, \{N_1, N_3, E_3, D_3\}\rangle.$

Finally, rule *twoloop* is applied to $\sigma'$ to remove node $N_1$, resulting in $\sigma''$:

$\sigma'' = \langle \mathrm{node}(N_3, D_3) \uplus \mathrm{edge}(E', N_3, N_3), \top, \{N_3, D_3\}\rangle.$

As can be seen the built-in store may contain a chain of degree adjustments for nodes with initially uninstantiated degree, although in this example it is not the case as all degrees remain unchanged. The other interesting consequences of partially uninstantiated encodings are investigated more thoroughly in Section 3.3.

### *3.2 Formal properties*

This section examines formal properties of the encoding given in Section 3.1. First, Section 3.2.1 analyses the special CHR states found when working with a GTS–CHR program. Then we prove soundness and completeness of the encoding in Section 3.2.2.

Our encoding is based on the assumption that the resulting CHR programs are executed only for initial states that correspond to graphs. We are not interested in executions for arbitrary CHR states.

#### *3.2.1 States encoding graphs*

In this section, we compare the different equivalence notions, i.e. graph isomorphism and CHR state equivalence, and present a formal characterization of a CHR state $\sigma$ that is the encoding of a graph $G$.

In order to determine if a CHR state encodes a graph, we define a predicate that holds if and only if this is the case. It is intuitively clear that starting with the encoding of a graph and transforming it via a graph transformation rule yields the

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

encoding of a graph again. Formally, this is an invariant according to the following definition. The first appearance of invariants in CHR research is found in Lam and Sulzmann (2006) in the context of agent programming.

*Definition 3.5* (*Invariant*)
An *invariant* $\mathscr{I}$ is a predicate such that for all $\sigma_0$ and $\sigma_1$, we have that if $\sigma_0 \rightarrowtail \sigma_1$ (or $\sigma_0 \equiv \sigma_1$) and $\mathscr{I}(\sigma_0)$ then $\mathscr{I}(\sigma_1)$.

The definition below introduces our desired property for CHR states. Note that it is referred to as an invariant here, although we do not require it to be an invariant throughout this section. In Section 3.2.2, more precisely Corollary 6, we will show that it is indeed a proper invariant.

*Definition 3.6* (*Graph invariant*)
Let $\sigma = \langle \mathbb{G}, \mathbb{B}_c \wedge \mathbb{B}_a, \mathbb{V} \rangle$ be a state, where $\mathbb{B}_c$ are constraints of the form $X = c$ for constants $c$ and $\mathbb{B}_a$ are constraints of the form $X = Y + c_1 - c_2$ for constants $c_1, c_2$.

The *graph invariant* $\mathscr{G}$ holds for state $\sigma$ if and only if there exists a graph $G$ and a conjunction $B$ of equality constraints of the form $X = c$ for a variable $X$ and constant $c$, such that

$$\langle \mathbb{G}, \mathbb{B}_c \wedge \mathbb{B}_a \wedge B, \emptyset \rangle \equiv \langle \mathtt{chr}(\mathrm{ground}, G), \top, \emptyset \rangle$$

For a state $\sigma$ for which $\mathscr{G}(\sigma)$ holds with a graph $G$ we say $\sigma$ is a *$\mathscr{G}$-state based on $G$*.

*Example 3.4*
Consider again the final state $\sigma''$ from the example computation in Section 3.1.1:

$$\sigma'' = \langle \mathrm{node}(N_3, D_3) \uplus \mathrm{edge}(E', N_3, N_3), \top, \{N_3, D_3\} \rangle.$$

By using the equality constraint $B = (D_3 = 2)$ the resulting state for Definition 3.6 is equivalent to:

$$\langle \mathrm{node}(N_3, 2) \uplus \mathrm{edge}(E', N_3, N_3), \top, \emptyset \rangle.$$

Let $G$ be the graph consisting of a node $v$ with a loop, then $\mathtt{chr}(\mathrm{ground}, G) = \mathrm{node}(N_v, 2) \uplus \mathrm{edge}(\tilde{E}, N_v, N_v)$. Therefore, the invariant $\mathscr{G}$ is satisfied for the above state $\sigma''$ as the corresponding states are equivalent by renaming of local variables.

This example further shows why the variable set $\mathbb{V}$ is disregarded for the two states. The variable given by var for a node of the graph has to coincide with the corresponding global variable for both states to be equivalent. Hence, for the above graph with node $v$, knowledge of the state $\sigma''$ would be necessary to determine that $\mathrm{var}(v) = N_3$. Omitting global variables from both states, however, allows us to freely map $v$ to any variable through $\mathrm{var}(v)$.

*Example 3.5*
For the state $\sigma = \langle \mathtt{chr}(\mathrm{keep}, G), \top, \mathbb{V} \rangle$ there clearly exists such a graph $G$, for which $B$ simply assigns the corresponding degree variables. States may also be in-between $\mathtt{chr}(\mathrm{ground}, G)$ and $\mathtt{chr}(\mathrm{keep}, G)$ in the sense that only some of the degree variables are instantiated, resulting in a state $\sigma' = \langle \mathtt{chr}(\mathrm{keep}, G), \mathbb{B}_c, \mathbb{V} \rangle$ with $\mathbb{B}_c$ being the corresponding equality constraints. By instantiating the remaining degrees it is clear that $\mathscr{G}(\sigma')$ holds.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

Note that arithmetic built-in constraints, introduced by bodies of rules in order to adjust a node's degree, are covered by the above graph invariant definition: The introduction of the corresponding degree equality constraint leads to a collapse of the chain of arithmetic constraints. Hence, the concept of a $\mathscr{G}$-state based on $G$ also applies to intermediate computation states, which gives rise to the following lemma.

*Lemma 3* (*Graph states*)
Let $\mathscr{G}(\sigma)$ hold for a state $\sigma$, then there exists a graph $G$ such that

$$\sigma \equiv \langle \mathtt{chr}(\mathrm{keep}, G), \mathbb{B}_c \wedge \mathbb{B}_a, \mathbb{V} \rangle$$

- $\mathbb{B}_c$ is a conjunction of $\mathrm{dvar}(v) = \deg_G(v)$ constraints
- $\mathbb{B}_a$ is a conjunction of $\mathrm{dvar}(v') = \mathrm{dvar}(v) + c_1 - c_2$ constraints.

*Proof*
Let $\sigma = \langle \mathbb{G}, \mathbb{B}_c \wedge \mathbb{B}_a, \mathbb{V} \rangle$, then by Definition 3.6 we have that $\langle \mathbb{G}, \mathbb{B}_c \wedge \mathbb{B}_a \wedge B, \emptyset \rangle \equiv \langle \mathtt{chr}(\mathrm{ground}, G), \top, \emptyset \rangle$ for a graph $G$ and $X = k$ constraints $B$.

W.l.o.g. all identifier variables occurring in $\mathtt{chr}(\mathrm{ground}, G)$ (and therefore in $\mathtt{chr}(\mathrm{keep}, G)$) also occur in $\mathbb{G}$ as identifier variables. Due to the state equivalence the difference between $\mathbb{G}$ and $\mathtt{chr}(\mathrm{keep}, G)$ can then only consist in $\mathbb{G}$ specifying some node degrees by constants (for degree variables we can again assume that they are the same as in $\mathtt{chr}(\mathrm{keep}, G)$).

Let $\Theta$ be a conjunction of equality constraints of the form $X = c$ for each degree specified explicitly in $\mathbb{G}$, using fresh variables for $X$. Interpreting $\Theta$ as a substitution, replacing $X$ with $c$ for each of the equivalences, we have that

$$\sigma \equiv \langle \mathtt{chr}(\mathrm{keep}, G)\Theta, \mathbb{B}_c \wedge \mathbb{B}_a, \mathbb{V} \rangle.$$

As all variables occurring in $\Theta$ are local, we get by Definition 2.8:

$$\begin{aligned} \sigma &\overset{\mathscr{CT}}{\equiv} \langle \mathtt{chr}(\mathrm{keep}, G)\Theta, \mathbb{B}_c \wedge \mathbb{B}_a \wedge \Theta, \mathbb{V} \rangle \\ &\overset{\text{Subst}}{\equiv} \langle \mathtt{chr}(\mathrm{keep}, G), \mathbb{B}_c \wedge \mathbb{B}_a \wedge \Theta, \mathbb{V} \rangle \\ &= \langle \mathtt{chr}(\mathrm{keep}, G), \mathbb{B}'_c \wedge \mathbb{B}_a, \mathbb{V} \rangle. \quad \square \end{aligned}$$

The reverse direction of Lemma 3 does not hold in general: The state $\sigma = \langle \emptyset, D = 0 \wedge X = 1 \wedge D = X + 2 - 0, \emptyset \rangle$ satisfies the conditions for an empty graph $G$, but of course $\mathscr{G}(\sigma)$ does not hold, as $\langle \emptyset, \bot, \emptyset \rangle \not\equiv \langle \emptyset, \top, \emptyset \rangle$.

The following lemma presents an interesting fact of the correspondence between state equivalence and graph isomorphism: equivalent CHR states encoding two graphs imply that these graphs are isomorphic.

*Lemma 4* (*Equivalent $\mathscr{G}$-states imply Graph Isomorphism*)
Given a state $\sigma_1 = \langle \mathtt{chr}(\mathrm{keep}, G_1), \mathbb{B}_1, \mathbb{V} \rangle$, a $\mathscr{G}$-state based on $G_1$, and a state $\sigma_2 = \langle \mathtt{chr}(\mathrm{keep}, G_2), \mathbb{B}_2, \mathbb{V} \rangle$, a $\mathscr{G}$-state based on $G_2$, then

$$\sigma_1 \equiv \sigma_2 \Rightarrow G_1 \simeq G_2$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

*Proof*
First, we note that $\mathbb{B}_1, \mathbb{B}_2$ consist only of degree equalities or adjustments. Therefore, we consider the following states instead, which are already sufficient to imply the isomorphism:

$$\langle \mathtt{chr}(\mathrm{keep}, G_1), \top, \mathbb{V} \rangle \equiv \langle \mathtt{chr}(\mathrm{keep}, G_2), \top, \mathbb{V} \rangle.$$

W.l.o.g. let the local variables occurring in the two states be disjoint (it is clear that otherwise we can consider equivalent states that only differ by renaming of local variables and that these states all provide corresponding graph isomorphisms).

Let $\bar{y}_1$ and $\bar{y}_2$ be the set of local variables of the two states. We can then apply the criterion from Theorem 2 to get

$$\mathscr{CT} \models \exists \bar{y}_1.\mathtt{chr}(\mathrm{keep}, G_1) = \mathtt{chr}(\mathrm{keep}, G_2).$$

As there are only variable terms contained in this equivalence we have the following conclusion, where $c(\bar{t})$ is any constraint with argument terms, i.e. variables, $\bar{t}$.

$$\exists f : \bar{y}_1 \to \bar{y}_2 \text{ with } c(\bar{t}) \in \mathtt{chr}(\mathrm{keep}, G_1) \to c(f(\bar{t})) \in \mathtt{chr}(\mathrm{keep}, G_2).$$

We know that $f$ is surjective (as the variables are disjoint and the above equality demands that at least one variable from $\bar{y}_1$ is mapped to each variable in $\bar{y}_2$). A consequence of this is that $|\bar{y}_1| \geqslant |\bar{y}_2|$.

Analogously, we get from $\mathscr{CT} \models \exists y_2.\mathtt{chr}(\mathrm{keep}, G_1) = \mathtt{chr}(\mathrm{keep}, G_2)$ that $|\bar{y}_2| \geqslant |\bar{y}_1|$, and hence, $|\bar{y}_1| = |\bar{y}_2|$. From this follows that $f$ is also injective, and therefore, bijective.

Next we realize that by the above equality, $f$ has to map local variables corresponding to node identifiers to local variables that also correspond to node identifiers. Let $\bar{y}_{n1} \subset \bar{y}_1, \bar{y}_{n2} \subset \bar{y}_2$ be the local variables used for node identifiers, then $f' : \bar{y}_{n1} \to \bar{y}_{n2}, y \mapsto f(y)$ is a well-defined and bijective function. We use this to define the actual graph isomorphism function $g : V_{G_1} \to V_{G_2}$:

$$g(v) = \begin{cases} v & \text{if } \mathrm{var}(v) \in \mathbb{V}, \\ v' & \text{if } \mathrm{var}(v) \in \bar{y}_{n1} \quad \text{and} \quad f'(\mathrm{var}(v)) = \mathrm{var}(v'), \end{cases}$$

$g$ is well defined: for every node there is a corresponding node identifier variable and it has to be either global or local. If it is local, then $f'$ has to map it to another local variable, as otherwise the $\equiv$ relation cannot hold. Furthermore, $g$ is bijective as well, because it is defined bijectively via $f'$ on local variables and the identity function on global variables.

Finally, $g$ is a graph isomorphism: By the above equality we have corresponding pairs of edge constraints. For every edge adjacent to a node given by a global variable, the corresponding edge has to be adjacent to the same node with the same global variable in order to satisfy $\equiv$. If the edge is adjacent to a node identified by a local variable, then this variable is bijectively mapped to another local variable and the above equality ensures that the corresponding edge is adjacent to the same node as well. □

The reverse direction of Lemma 4 cannot hold in general: The encoding of the graphs $G_1$ and $G_2$ are independent from determining the set $\mathbb{V}$ of global variables.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

Even a graph consisting of a single node only can be encoded in two ways, such that the states are not equivalent:

$$\langle \text{node}(N,0), \top, \emptyset \rangle \not\equiv \langle \text{node}(N,0), \top, \{N\} \rangle.$$

As indicated in Section 3.1.1, states may contain node encodings with a variable degree. As these states are fundamental for program analysis the following definition characterizes the set of these nodes.

*Definition 3.7* (*Strong nodes*)
For a CHR state $\sigma \equiv \langle \texttt{chr}(\text{keep}, G), \mathbb{B}, \mathbb{V} \rangle$ which is a $\mathscr{G}$-state based on $G$ we define the set of *strong nodes* as

$$\mathscr{S}(\sigma) = \{v \in V_G \mid \text{dvar}(v) = \deg_G(v) \notin \mathbb{B}\}.$$

The effect of strong nodes on computations and their use in program analysis is discussed in detail in Section 3.3.

### 3.2.2 Soundness and completeness

In this section, we prove soundness and completeness of our embedding. That $\mathscr{G}$ is an invariant for a GTS–CHR program and that termination of a GTS and its GTS–CHR program coincide, are then derived as consequences of the main theorem below.

*Theorem 5* (*Soundness and completeness*)
Let $\sigma \equiv \langle \texttt{chr}(\text{keep}, G), \mathbb{B}, \mathbb{V} \rangle$ be a CHR state with $\mathscr{G}(\sigma)$ holding with graph $G$. Then

$$G \overset{r,m}{\Longrightarrow} H \text{ with } \{v \in V_G \mid \text{tr}_{G \Rightarrow H}(v) \text{ defined}\} \supseteq \mathscr{S}(\sigma)$$

if and only if

$$\sigma \rightarrowtail^{r} \tau \equiv \langle \texttt{chr}(\text{keep}, H), \mathbb{B}', \mathbb{V} \rangle \text{ and } \mathscr{G}(\tau) \text{ holds with graph } H.$$

*Proof*
In order to shorten this proof we use $k(G)$ and $g(G)$ to denote $\texttt{chr}(\text{keep}, G)$ and $\texttt{chr}(\text{ground}, G)$, respectively.

"$\Longrightarrow$":

Let $G \overset{r,m}{\Longrightarrow} H$ and let $r : L \leftarrow K \rightarrow R$.

Let $\mathbb{G} := k(G) = k(G \backslash m(L)) \uplus k(m(E_L)) \uplus k(m(V_K)) \uplus k(m(V_L \backslash V_K)) \Rightarrow \sigma \equiv \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$.

Let $\varrho(r) = (r \mathbin{@} C_L \Leftrightarrow C_R^b, C_R^u)$ with $C_L = k(K) \uplus g(L \setminus K)$.

For $v \in V_L$ we have $\text{type}_G(v)(\text{var}(v), _) \in C_L$ and $\text{type}_G(v)(\text{var}(m(v)), \text{dvar}(m(v))) \in k(m(V_K))$, as the types match due to $m$ being a graph morphism.

As we have a fresh rule using node $v$ that does not occur elsewhere we can say that $\sigma \overset{\mathscr{C},\mathscr{T}}{\equiv} \langle \mathbb{G}, \text{var}(m(v)) = \text{var}(v) \wedge \mathbb{B}, \mathbb{V} \rangle$, and hence

$$\sigma \overset{\text{Subst}}{\equiv} \langle \mathbb{G}[\text{var}(m(v))/\,\text{var}(v)], \text{var}(m(v)) = \text{var}(v) \wedge \mathbb{B}, \mathbb{V} \rangle. \quad (1)$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

Consider $v \in V_L \setminus V_K$, then $\text{type}_G(v)(\text{var}(v), \deg_L(v)) \in C_L$. Assume that $m(v) \in \mathscr{S}(\sigma)$, then $\text{tr}_{G \Rightarrow H}(m(v))$ is defined, which is a contradiction to $v \in V_L \setminus V_K$. Therefore, $m(v) \notin \mathscr{S}(\sigma)$ and hence $\text{dvar}(m(v)) = \deg_G(m(v)) \in \mathbb{B}$. As $G \overset{r,m}{\Longrightarrow} H$ satisfies the gluing condition, we know that $\deg_L(v) = \deg_G(m(v))$. Therefore, we have that

$$\sigma \overset{\text{Subst}}{\equiv} \langle \mathbb{G}[\text{var}(m(v))/\text{var}(v)][\text{dvar}(m(v))/\deg_G(m(v))], \text{var}(m(v)) = \text{var}(v) \wedge \mathbb{B}, \mathbb{V} \rangle.$$

From (1) for nodes $v \in V_K$ and the above for nodes $v \in V_L \setminus V_K$ follows for a conjunction of equality constraints $E$ that

$$\sigma \equiv \langle k(G \setminus m(L)) \uplus k(m(E_L)) \uplus k(V_K) \uplus g(V_L \setminus V_K), \mathbb{B} \wedge E, \mathbb{V} \rangle = \langle \mathbb{G}', \mathbb{B} \wedge E, \mathbb{V} \rangle.$$

Let $e \in E_L$, than $\text{type}_G(e)(\text{var}(e), \text{var}(\text{src}(e)), \text{var}(\text{tgt}(e))) \in C_L$ and after the previous substitutions have been made for node identifier variables, and as $k(e) = g(e)$, we get $\text{type}_G(m(e))(\text{var}(m(e)), \text{var}(\text{src}(e)), \text{var}(\text{tgt}(e))) \in \sigma$ . We then have

$$\sigma \equiv \langle \mathbb{G}'[\text{var}(m(e))/\text{var}(e)], \text{var}(m(e)) = \text{var}(e) \wedge \mathbb{B} \wedge E, \mathbb{V} \rangle. \tag{2}$$

By applying this substitution for all edges $e \in E_L$ and extending $E$ with the required equalities to $E'$ we get:

$$\sigma \equiv \langle k(G \setminus m(L)) \uplus k(E_K) \uplus g(E_L \setminus E_K) \uplus k(V_K) \uplus g(V_L \setminus V_K), \mathbb{B} \wedge E', \mathbb{V} \rangle.$$

Hence, $\sigma \equiv \langle k(G \setminus m(L)) \uplus C_L, \mathbb{B} \wedge E', \mathbb{V} \rangle$ such that we apply the rule $\varrho(r)$ to $\sigma$:

$$\begin{aligned} \sigma \rightarrowtail^r \tau &\equiv \langle k(G \setminus m(L)) \uplus C_R^u, \mathbb{B} \wedge E' \wedge C_R^b, \mathbb{V} \rangle \\ &\equiv \langle k(G \setminus m(L)) \uplus g(R \setminus K) \uplus k(E_K) \uplus k(V_K'), \mathbb{B} \wedge E' \wedge C_R^b, \mathbb{V} \rangle. \end{aligned}$$

As $C_R^b$ contains $\text{var}(v') = \text{var}(v) \forall v \in V_K$ let $C_R'$ be $C_R^b$ without these constraints, then

$$\begin{aligned} \tau &\overset{\text{Subst}}{\equiv} \langle k(G \setminus m(L)) \uplus g(R \setminus K) \uplus k(E_K) \uplus k(V_K), \mathbb{B} \wedge E' \wedge C_R^b, \mathbb{V} \rangle \\ &\overset{\mathscr{CT}}{\equiv} \langle k(G \setminus m(L)) \uplus g(R \setminus K) \uplus k(E_K) \uplus k(V_K), \mathbb{B} \wedge E' \wedge C_R', \mathbb{V} \rangle. \end{aligned}$$

Let $\hat{\mathbb{G}} := k(G \setminus m(L)) \uplus k(K)$, then $\tau \overset{\mathscr{CT}}{\equiv} \langle \hat{\mathbb{G}} \uplus g(R \setminus K), \mathbb{B} \wedge E' \wedge C_R' \wedge D_R, \mathbb{V} \rangle$ with $\forall v \in V_R \setminus V_K . \text{dvar}(v) = \deg_R(v) \in D_R$. Furthermore, consider $\Theta$ a substitution corresponding to the reverse reading of $E'$ which undoes the ideas of (1) and (2) for all affected nodes and edges. We then get

$$\begin{aligned} \tau &\overset{\text{Subst}}{\equiv} \langle \hat{\mathbb{G}} \uplus k(R \setminus K), D_R \wedge C_R' \wedge \mathbb{B} \wedge E', \mathbb{V} \rangle \\ &\overset{\text{Subst}}{\equiv} \langle k(G \setminus m(L \setminus K)) \uplus (k(R \setminus K)\Theta), D_R \wedge C_R' \wedge \mathbb{B} \wedge E', \mathbb{V} \rangle \\ &\overset{\mathscr{CT}}{\equiv} \langle k(G \setminus m(L \setminus K)) \uplus (k(R \setminus K)\Theta), D_R \wedge C_R'\Theta \wedge \mathbb{B}, \mathbb{V} \rangle \\ &\equiv \langle k(H), \mathbb{B}', \mathbb{V} \rangle \end{aligned}$$

We get the graph $H$ as its DPO construction corresponds to the removal of $m(L \setminus K)$ and addition of $R \setminus K$. $\Theta$ is needed to attach the new nodes of $R \setminus K$ to nodes from $V_K$ and $C_R'$ contains degree adjustments for those nodes that are correct by construction. Hence, it also holds that $\mathscr{G}(\tau)$ is satisfied with graph $H$.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

"$\Longleftarrow$":

Let $\sigma \rightarrowtail^r \tau$ with $\tau \equiv \langle k(H), \mathbb{B}', \mathbb{V}\rangle$ and $\mathscr{G}(\tau)$ holds with graph $H$. Let $\varrho(r) = (r \ @ \ C_L \Leftrightarrow C_R^b, C_R^u$ with $C_L = k(K) \uplus g(L \setminus K)$. From Definition 2.10 follows that

$$\sigma \equiv \langle k(K) \uplus g(L \setminus K) \uplus k(G \setminus L), \mathbb{B}_1, \mathbb{V}\rangle \tag{3}$$

Using Lemma 3 and with $E$ being a conjunction of $\mathrm{var}(m(x)) = \mathrm{var}(x)$ constraints for $x \in L$ we get:

$$\begin{aligned}\sigma &\equiv \langle k(G), \mathbb{B}_c \wedge \mathbb{B}_a, \mathbb{V}\rangle \\ &\equiv \langle k(K) \uplus k(L \setminus K) \uplus k(G \setminus m(L)), \mathbb{B}_c \wedge \mathbb{B}_a \wedge E, \mathbb{V}\rangle \\ &\overset{(3)}{\equiv} \langle k(K) \uplus g(L \setminus K) \uplus k(G \setminus m(L)), \mathbb{B}_c \wedge \mathbb{B}_a \wedge E', \mathbb{V}\rangle,\end{aligned}$$

where $E'$ is the extension of $E$ with $\mathrm{dvar}(m(v)) = \deg_G(v)$ constraints for $v \in V_L \setminus V_K$ and $\mathbb{B}_1 = \mathbb{B}_c \wedge \mathbb{B}_a \wedge E'$.

$m : L \to G$ is well defined and injective by the multiset semantics of CHR and it remains to be shown that $m$ is a graph morphism. Therefore, let $e \in E_L$, then $\mathrm{type}_L(e)(\mathrm{var}(e), \mathrm{var}(\mathrm{src}(e)), \mathrm{var}(\mathrm{tgt}(e))) \in C_L$ and $\mathrm{type}_L(\mathrm{src}(e))(\mathrm{var}(\mathrm{src}(e)), _) \uplus \mathrm{type}_L(\mathrm{tgt}(e))(\mathrm{var}(\mathrm{tgt}(e)), _) \in C_L$. Hence, $\mathrm{var}(m(e)) = \mathrm{var}(e)$, $\mathrm{var}(m(\mathrm{src}(e))) = \mathrm{var}(\mathrm{src}(e))$ and $\mathrm{var}(m(\mathrm{tgt}(e))) = \mathrm{var}(\mathrm{tgt}(e))$ are all in $\mathbb{B}_1$. Therefore, $m(\mathrm{src}(e)) = \mathrm{src}(m(e)) \wedge m(\mathrm{tgt}(e)) = \mathrm{tgt}(m(e))$.

The gluing condition is satisfied, as $\forall v \in V_L \setminus V_K$ the matched degree ensures that there are no dangling edges, hence, $r$ is GTS-applicable to $G$. Similarly to the other proof direction, we show that the DPO construction of $H$ coincides with the construction of $\tau$ by CHR rule application:

$$\begin{aligned}\sigma \rightarrowtail^r \tau \ &\equiv \ \langle k(K) \uplus g(R \setminus K) \uplus k(G \setminus m(L)), \mathbb{B}_c \wedge \mathbb{B}_a \wedge E' \wedge C_R^b, \mathbb{V}\rangle \\ &\overset{\mathscr{CT}}{\equiv} \ \langle k(K) \uplus g(R \setminus K) \uplus k(G \setminus m(L)), \mathbb{B}_c \wedge \mathbb{B}_a \wedge E \wedge C_R^b, \mathbb{V}\rangle \\ &\overset{\text{Subst}}{\equiv} \ \langle k(m(K)) \uplus g(R \setminus K) \uplus k(G \setminus m(L)), \mathbb{B}_c \wedge \mathbb{B}_a \wedge E \wedge C_R^b, \mathbb{V}\rangle \\ &= \ \langle g(R \setminus K) \uplus k(G \setminus m(L \setminus K)), \mathbb{B}_c \wedge \mathbb{B}_a \wedge E \wedge C_R^b, \mathbb{V}\rangle \\ &\equiv \ \langle g(R \setminus K)\Theta \uplus k(G \setminus m(L \setminus K)), \mathbb{B}_c \wedge \mathbb{B}_a \wedge C_R^b, \mathbb{V}\rangle \\ &\equiv \ \langle k(H), \mathbb{B}', \mathbb{V}\rangle\end{aligned}$$

where $\Theta$ is the reverse substitution for $E$ similar to the other proof direction. The final equivalence comes from extracting the degrees of constraints in $g(R \setminus K)$ into equality constraints contained in $\mathbb{B}'$. As can be seen here, the application of the rule results in a state encoding the graph $H$, such that $\mathscr{G}(\tau)$ holds.

Finally, for the set $\mathscr{S}(\sigma)$ we know that the nodes cannot be removed by rule $r$: For a node $v \in V_L \setminus V_K$ we have $\mathrm{type}_L(v)(\mathrm{var}(v), \deg_L(v)) \in C_L$, but this cannot be matched with $\sigma$, as by Definition 3.7 the corresponding degree is unavailable. Hence, none of the nodes from $\mathscr{S}(\sigma)$ are removed by the rule application $G \overset{r,m}{\Longrightarrow} H$, i.e. $\mathrm{tr}_{G \Rightarrow H}(v)$ is defined for all $v \in \mathscr{S}(\sigma)$. □

As can be seen in the proof of Theorem 5, a GTS–CHR rule application on a $\mathscr{G}$-state based on $G$ always results in a state encoding a corresponding graph $H$, which gives us the following corollary.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

*Corollary 6* (*$\mathscr{G}$ Invariant*)
For a GTS–CHR program $\mathscr{G}$ is an invariant.

A closer look at the conditions required in Theorem 5 reveals that for a state $\sigma$ with $\mathscr{S}(\sigma) = \emptyset$, i.e. for an encoding of a graph with all degrees explicitly given, we have unrestricted soundness and completeness.

*Corollary 7* (*Unrestricted Soundness and Completeness*)
Let $\sigma \equiv \langle \mathtt{chr}(\mathrm{ground}, G), \top, \mathbb{V} \rangle$ be a CHR state with $\mathscr{G}(\sigma)$ holding with graph $G$. Then

$$G \overset{r,m}{\Longrightarrow} H$$

if and only if

$$\sigma \rightarrowtail^{r} \tau \equiv \langle \mathtt{chr}(\mathrm{ground}, H), \top, \mathbb{V} \rangle \text{ and } \mathscr{G}(\tau) \text{ holds with graph } H$$

*Proof*
This follows from Theorem 5 and the following insight: as all degrees of $G$ are specified explicitly and all nodes added by the rule are also given explicit degrees, all degrees in $H$ are given explicitly as well, which allows us to use $\mathtt{chr}(\mathrm{ground}, H)$ here. □

Finally, the soundness and completeness result induces a termination correspondence between a GTS and its GTS–CHR program. Again, we restrict our observation to graph-encoding states.

*Corollary 8* (*Termination Correspondence*)
A GTS is terminating if and only if its corresponding GTS–CHR program is $\mathscr{G}$-terminating, i.e. terminating for all $\mathscr{G}$-states.

*Proof*
If a GTS contains a non-terminating derivation, we have the corresponding computation in its GTS–CHR program by Corollary 7. Similarly, if the GTS–CHR program has a non-terminating computation, there exists a corresponding non-terminating GTS derivation according to Theorem 5. □

### *3.3 Discussion*

In this section we discuss our previously presented encoding. First, Section 3.3.1 investigates that a GTS–CHR program works with partially defined graphs and explains the suitability of these graphs for program analysis. Then we present ways to simplify the encoding of GTS–CHR rules in Section 3.3.2.

#### *3.3.1 Partially defined graphs*

In the example computation given in Section 3.1.1 the input contains a node with a variable degree: $\mathrm{node}(N_3, D_3)$. Nevertheless, computations on this input are possible and the example resulted in the final state:

$$\langle \mathrm{node}(N_3, D_3) \uplus \mathrm{edge}(E', N_3, N_3), \top, \{N_3, D_3\} \rangle$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

In general, a variable node degree will cause a chain of degree adjustment constraints to be created, i.e. constraints of the form $X = Y + c_1 - c_2$. These stem from the node being involved in a rule application that affects its degree.

It is important to realize that we can only match such a node in rules that do not remove it. A rule that removes a node contains the explicit degree for that node in the head, which cannot be matched through a variable degree. As a consequence, specifying variable degrees in the input ensures that the corresponding nodes will not be removed by the computation. This also becomes clear from the investigation of strong nodes in the previous section.

While this is an interesting feature in its own right, it provides the basis for many forms of program analysis. The aim of program analysis is to make statements on an infinite number of graphs, while only having to investigate a small selection of graphs. Graph encodings with variable degrees can here be thought of as partially defined graphs, i.e. there may be any number of further edges being connected to a node with a variable degree.

Note that partially defined graphs only exist within the CHR context. In a GTS the degree of a node is implicitly given by the adjacent edges. As a consequence, leaving a node's degree undefined in the CHR encoding ensures that this node will not be removed during computation. In the GTS context we have no such option available for host graphs.

By the above argument, the state

$$\langle \mathrm{node}(N, D), \top, \{N, D\} \rangle$$

therefore not only represents the graph consisting of a single node and no edges. Instead, it represents the set of all graphs with at least one node. Similarly, the above final state from Section 3.1.1 stands for the set of graphs that contain at least one node with a loop.

Every computation performed on an input with variable degrees actually represents computations for an infinite set of graphs. This is a fundamental feature for the usage of our encoding in program analysis and will be exploited in Sections 4 and 5.

### *3.3.2 Different encoding possibilities*

The encoding proposed in this work can be varied in several different ways. We chose the encoding in Definitions 3.2 and 3.3 for this work, because it is a verbose encoding, hence, directly presenting all its components and simplifying the proofs. In practice however, a less verbose encoding resulting in shorter rules can be used instead. In this section we present different possible simplifications achieving this.

The different simplifications are illustrated by applying them to the *twoloop* rule which is of the following form when encoded as specified in Definition 3.3:

$$\begin{aligned} \text{twoloop} \ @ \quad & \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N_2, 2) \uplus \\ & \mathrm{edge}(E_1, N_1, N_2) \uplus \mathrm{edge}(E_2, N_2, N_1) \\ & \Leftrightarrow \\ & \mathrm{node}(N_1', D_1') \uplus \mathrm{edge}(E_3, N_1, N_1), N_1' = N_1 \wedge D_1' = D_1 - 2 + 2 \end{aligned}$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

There are two ways to specify the degree of nodes in $L \setminus K$. The one chosen in Definition 3.3 explicitly specifies the respective degree in the head. Another way is to keep the degree as a variable $D$ in the head and add the built-in constraint $D = k$ to the guard of the rule. However, most current CHR compilers detect these equalities and automatically transform between them to the representation most suitable for an optimization. Therefore, in this work we directly specify the degree in the head to avoid guards altogether.

*Variable elimination* As Definition 3.3 encodes a node $v \in V_K$ using a new node identifier $v'$ with $\mathrm{var}(v) = \mathrm{var}(v')$ and $\mathrm{var}(v')$ is not used elsewhere, this substitution can be included directly into the rule encoding:

$$\begin{aligned} \text{twoloop } @ \quad & \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N_2, 2) \uplus \\ & \mathrm{edge}(E_1, N_1, N_2) \uplus \mathrm{edge}(E_2, N_2, N_1) \\ & \Leftrightarrow \\ & \mathrm{node}(N_1, D_1') \uplus \mathrm{edge}(E_3, N_1, N_1), D_1' = D_1 - 2 + 2 \end{aligned}$$

Note that we perform variable elimination on node identifiers by default in the remainder of this work. However, as we need to take degree adjustments into account, the formulation of Definition 3.3 is simplified by the variable duplication.

*Arithmetic simplification* The degree adjustments in Definition 3.3 explicitly contain the information on how many edges the rule deletes and creates. For the adjustment itself, however, it is sufficient to simply adjust the degree by the actual change in the number of edges. Additionally, if the change is 0, like in the *twoloop* rule, the extra local variable used for the degree can be substituted, resulting in:

$$\begin{aligned} \text{twoloop } @ \quad & \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N_2, 2) \uplus \\ & \mathrm{edge}(E_1, N_1, N_2) \uplus \mathrm{edge}(E_2, N_2, N_1) \\ & \Leftrightarrow \\ & \mathrm{node}(N_1, D_1) \uplus \mathrm{edge}(E_3, N_1, N_1) \end{aligned}$$

*Elimination of Edge Identifiers* The edge identifier variables are used throughout this work, because they simplify dealing with the multiset semantics of CHR with respect to the edge constraint representing exactly one edge of a graph. In a CHR implementation, however, every constraint is implemented as a unique object – sometimes even annotated with an identifier number – which makes the explicit edge identifiers redundant. Using this idea the *twoloop* rule can be further simplified to:

$$\begin{aligned} \text{twoloop } @ \quad & \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N_2, 2) \uplus \\ & \mathrm{edge}(N_1, N_2) \uplus \mathrm{edge}(N_2, N_1) \\ & \Leftrightarrow \\ & \mathrm{node}(N_1, D_1) \uplus \mathrm{edge}(N_1, N_1) \end{aligned}$$

Note that the same argumentation cannot be applied to node identifiers, as those are required for specifying the source and target of edge constraints.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

*Simpagation Rules* Some nodes and edges of the left-hand rule graph $L$ of a GTS rule can occur only to specify a certain graph context and are unaffected by the rule application. For nodes this can also happen if the modification to adjacent edges results in no change to the degree, as in the *twoloop* rule. In those cases, the node or edge is encoded in exactly the same way in the head and body of the rule. Therefore, during the rule application the corresponding constraint is removed and introduced again. Using a simpagation rule allows us to move such a constraint into the part of the head which is not removed during the rule application. This reduces the textual size of the rule as well as its execution time, because it avoids the generation of a new constraint during the rule application.

After applying all the previous simplifications to the *twoloop* rule and transforming it into a simpagation rule we get the following simplified rule:

$$\begin{aligned} \text{twoloop} \ @ \quad & \text{node}(N_1, D_1) \backslash \\ & \text{node}(N_2, 2) \uplus \text{edge}(N_1, N_2) \uplus \text{edge}(N_2, N_1) \\ & \Leftrightarrow \\ & \text{edge}(N_1, N_1) \end{aligned}$$

One might be tempted to always create simpagation rules in Definition 3.3, based on the idea that the context graph $K$ already identifies non-removed nodes. However, the above creation of simpagation rules with node constraints among the kept constraints, is only possible if the respective node's degree remains unchanged by the rule application.

Readers more familiar with CHR may also wonder if propagation rules could be used as well. It is technically possible to define a GTS rule that does not remove any elements, but only adds new nodes and edges. However, a thusly created GTS would suffer from a problem that in CHR literatue is referred to as trivial non-termination (see e.g. Frühwirth 2009), i.e. such a rule could be applied infinitely often. For this reason, most CHR implementations restrict propagation rule applications, hence, our encoding using simplification or simpagation rules remains more faithful to the semantics of graph transformations.

## 4 Analysing confluence

The confluence property is relevant to both, GTS and CHR. It guarantees that any terminating computation made for an initial state results in the same final state no matter in which order applicable rules are applied.

In Section 4.1 we formally introduce confluence, both for GTS and CHR. Furthermore, we give the definitions for *critical pairs* in both systems, which are derived directly from the rules. Investigation of critical pairs for determining confluence of a terminating rewrite system goes back to research about term rewriting systems (Huet 1980), and both, GTS and CHR, have adapted the corresponding criteria.

Next, Section 4.2 examines the relation between critical pairs of a GTS and its corresponding GTS–CHR program. We then introduce the concept of *observable confluence* (Duck *et al*. 2007). It is a technical means to restrict our observations to

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

CHR states that correspond to graphs. This in turn results in a closer correspondence between GTS and CHR for later results.

For terminating GTS, confluence analysis proved to be undecidable: (Plump 2005) showed that the critical pair analysis gives only a sufficient criterion for confluence. We show that the decidable observable confluence test of a GTS–CHR program coincides with this criterion.

The discrepance in decidability of the two systems' confluence properties is discussed in Section 4.3 for exemplary critical pair analyses.

### *4.1 Preliminaries*

This subsection introduces the necessary definitions for GTS and CHR confluence before comparing the two notions. Unless noted otherwise, the involved GTS and GTS–CHR programs are assumed to be terminating.

*Definition 4.1* (*GTS Confluence*)
A GTS is called *confluent* if, for all typed graph transformations $G \overset{*}{\Longrightarrow} H_1$ and $G \overset{*}{\Longrightarrow} H_2$, there is a typed graph $X$ together with typed graph transformations $H_1 \overset{*}{\Longrightarrow} X$ and $H_2 \overset{*}{\Longrightarrow} X$. *Local confluence* means that this property holds for all pairs of direct typed graph transformations $G \Rightarrow H_1$ and $G \Rightarrow H_2$ (Ehrig *et al*. 2006).

Newman's general result for rewriting systems (Newman 1942) implies that it is sufficient to consider local confluence for terminating GTS. To verify local confluence, we particularly need to study critical pairs and their joinability, according to the following definition based on (Ehrig *et al*. 2006; Plump 2005).

*Definition 4.2* (*Joinability of Critical GTS Pair*)
Let $r_1 = (L_1 \overset{l}{\leftarrow} K_1 \overset{r}{\rightarrow} R_1), r_2 = (L_2 \overset{l}{\leftarrow} K_2 \overset{r}{\rightarrow} R_2)$ be two GTS rules. A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ 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 $m_1 : L_1 \to G$ and $m_2 : L_2 \to G$ is jointly surjective.

A pair $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ 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 $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ is called *joinable* if there exist typed graphs $X_1, X_2$ together with typed graph transformations $P_1 \overset{*}{\Longrightarrow} X_1 \simeq X_2 \overset{*}{\Longleftarrow} P_2$. It is *strongly joinable* if there is an isomorphism $f : X_1 \to X_2$ such that for each node $v$, for which $\mathrm{tr}_{G \Rightarrow P_1}(v)$ and $\mathrm{tr}_{G \Rightarrow P_2}(v)$ are defined, the following holds:

(1) $\mathrm{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)$ and $\mathrm{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$ are defined and
(2) $f_V(\mathrm{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)) = \mathrm{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$

A similar notion of confluence has been developed for CHR. The following definition is an adaptation of (Frühwirth 2009) to the operational semantics on equivalence classes.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

*Definition 4.3* (*CHR Confluence*)
A CHR program is called *confluent* if for all states $\sigma, \sigma_1$, and $\sigma_2$: If $\sigma_1 \ {}^*\!\leftarrowtail \sigma \rightarrowtail^* \sigma_2$, then $\sigma_1$ and $\sigma_2$ are joinable. Two states $\sigma_1$ and $\sigma_2$ are called *joinable* if there exists a state $\tau$ such that $\sigma_1 \rightarrowtail^* \tau \ {}^*\!\leftarrowtail \sigma_2$.

Analogous to a GTS, the confluence property for terminating CHR programs is determined by local confluence which can be checked through critical pairs. The following definition is adapted to the situation in this work, i.e. it only considers simplification rules and no guards.

*Definition 4.4* (*Joinability of Critical CHR Pair*)
Let $r_i, i = 1, 2$ be two (not necessarily different) simplification rules of the following kind with variables that have been renamed apart:

$$H_i \uplus A_i \Leftrightarrow B_i^u, B_i^b$$

Then an *overlap* $\sigma_{\mathscr{CP}}$ of $r_1$ and $r_2$ is $\sigma_{\mathscr{CP}} = \langle H_1 \uplus A_1 \uplus H_2, A_1 = A_2, \mathbb{V}\rangle$, provided $A_1$ and $A_2$ are non-empty multisets, $\mathbb{V} = \text{vars}(H_1 \uplus A_1 \uplus H_2 \uplus A_2)$ and $\mathscr{CT} \models \exists(A_1 = A_2)$.

Let $\sigma_1 = \langle B_1^u \uplus H_2, B_1^b \wedge (A_1 = A_2), \mathbb{V}\rangle$ and $\sigma_2 = \langle B_2^u \uplus H_1, B_2^b \wedge (A_1 = A_2), \mathbb{V}\rangle$. Then the tuple $\mathscr{CP} = (\sigma_1, \sigma_2)$ is a *critical CHR pair* of $r_1$ and $r_2$. A critical CHR pair $(\sigma_1, \sigma_2)$ is *joinable* if $\sigma_1$ and $\sigma_2$ are joinable.

### 4.2 Analysing confluence via critical pairs

After defining the different notions of confluence we now further investigate the difference between critical GTS pairs and critical CHR pairs for GTS–CHR programs. The following lemma shows that there exists a corresponding overlap for each critical GTS pair. Therefore, by examining the overlaps and using the previous soundness result we can transfer joinability results to the critical GTS pair.

*Lemma 9* (*Overlap for Critical GTS Pair*)
If $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ is a critical GTS pair, then there exists an overlap $\sigma_{\mathscr{CP}}$ of $\varrho(r_1) = (r_1 \ @ \ C_{L1} \Leftrightarrow C_{R1}^u, C_{R1}^b)$ and $\varrho(r_2) = (r_2 \ @ \ C_{L2} \Leftrightarrow C_{R2}^u, C_{R2}^b)$ which is a $\mathscr{G}$-state based on $G$ and a critical CHR pair $(\sigma_1, \sigma_2)$ such that $\sigma_1$ is a $\mathscr{G}$-state based on $P_1$ and $\sigma_2$ is a $\mathscr{G}$-state based on $P_2$.

*Proof*
Let the two GTS rules be $L_i \leftarrow K_i \rightarrow R_i$ for $i = 1, 2$ and let $M = m_1(L_1) \cap m_2(L_2)$. We then define the following sets of constraints from which we construct the overlap:

$$\begin{aligned}
H_1 &= \{\texttt{chr}_{L_1}(\text{keep}, x) \mid x \in L_1 \wedge m_1(x) \notin M\}\\
H_2 &= \{\texttt{chr}_{L_2}(\text{keep}, x) \mid x \in L_2 \wedge m_2(x) \notin M\}\\
A_1 &= \{\texttt{chr}_{L_1}(\text{keep}, x) \mid x \in L_1 \wedge m_1(x) \in M\}\\
A_2 &= \{\texttt{chr}_{L_2}(\text{keep}, x) \mid x \in L_2 \wedge m_2(x) \in M\}\\
C_1 &= \{\text{dvar}(v) = \deg_{L_1}(v) \mid v \in V_{L_1} \setminus V_{K_1}\}\\
C_2 &= \{\text{dvar}(v) = \deg_{L_2}(v) \mid v \in V_{L_2} \setminus V_{K_2}\}
\end{aligned}$$

Let $\mathbb{V} = \text{vars}(H_1 \uplus H_2 \uplus A_1 \uplus A_2)$ and let $\sigma = \langle H_1, C_1, \mathbb{V}\rangle$, then $\sigma \equiv \sigma' = \langle \{\texttt{chr}_{L_1}(\text{keep}, x) \mid x \in K_1 \wedge m_1(x) \notin M\} \uplus \{\texttt{chr}_{L_1}(\text{ground}, x) \mid x \in L_1 \setminus K_1 \wedge m_1(x) \notin$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

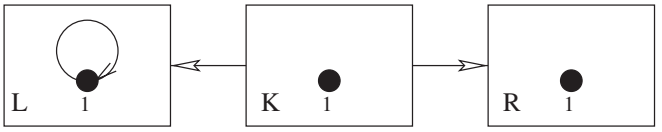



Fig. 10. Graph production rule for removing a loop.

$M\}, \top, \mathbb{V}\rangle =: \langle H_2', \top, \mathbb{V}\rangle$ by applying $C_1$ as a substitution to $H_1$, and then removing $C_1$ as all dvar($v$) variables for $v \in V_{L_1} \setminus V_{K_1}$ are then strictly local.

Similarly, $\langle A_1, C_1, \mathbb{V}\rangle \equiv \langle \{\mathtt{chr}(\mathrm{keep}, x) \mid x \in K_1 \wedge m_1(x) \in M\} \uplus \{\mathtt{chr}(\mathrm{ground}, x) \mid x \in L_1 \setminus K_1 \wedge m_1(x) \in M, \top, \mathbb{V}\rangle =: \langle A_1', \top, \mathbb{V}\rangle$, and analogously, we define $H_2'$ and $A_2'$.

By Definition 3.3 we have that $H_1' \uplus A_1' = C_{L1}$ and $H_2' \uplus A_2' = C_{L2}$. As $M \neq \emptyset$ it follows that $A_1'$ and $A_2'$ are non-empty. To investigate if $\mathscr{CT} \models \exists(A_1' = A_2')$ we take a closer look at the equality constraints imposed by $A_1' = A_2'$:

$$
\begin{aligned}
&\{\mathrm{var}(v_1) = \mathrm{var}(v_2) \mid v_1 \in V_{L_1} \wedge v_2 \in V_{L_2}, m_1(v_1) = m_2(v_2)\} \\
\wedge\ &\{\mathrm{dvar}(v_1) = \mathrm{dvar}(v_2) \mid v_1 \in V_{K_1} \wedge v_2 \in V_{K_2} \wedge m_1(v_1) = m_2(v_2)\} \\
\wedge\ &\{\mathrm{dvar}(v_1) = \deg_{L_2}(v_2) \mid v_1 \in V_{K_1} \wedge v_2 \in V_{L_2} \setminus V_{K_2} \wedge m_1(v_1) = m_2(v_2)\} \\
\wedge\ &\{\mathrm{dvar}(v_2) = \deg_{L_1}(v_1) \mid v_1 \in V_{L_1} \setminus V_{K_1} \wedge v_2 \in V_{K_2} \wedge m_1(v_1) = m_2(v_2)\} \\
\wedge\ &\{\mathrm{var}(e_1) = \mathrm{var}(e_2) \mid e_1 \in E_{K_1} \wedge e_2 \in E_{K_2} \wedge m_1(e_1) = m_2(e_2)\} \\
\wedge\ &\{\deg_{L_1}(v_1) = \deg_{L_2}(v_2) \mid v_1 \in V_{L_1} \setminus V_{K_1} \wedge v_2 \in V_{L_1} \setminus V_{K_2} \wedge \\
&\quad m_1(v_1) = m_2(v_2)\}
\end{aligned}
$$

Except for the last row, the above equality constraints can easily be satisfied under existential quantification. Hence, the only remaining problematic case is when two node constraints with constant degrees are overlapped. However, the degree of $m_1(v_1) = m_2(v_2)$ equals the degree of $v_1$ and the degree of $v_2$ due to the gluing condition being satisfied, such that this case can only occur with equal constant degrees.

Hence, $\sigma_{\mathscr{CP}} = \langle H_1' \uplus A_1' \uplus H_2', A_1' = A_2', \mathbb{V}\rangle$ is an overlap of $\varrho(r_1)$ and $\varrho(r_2)$ with the critical CHR pair $(\langle C_{R1}^u \uplus H_2', A_1' = A_2' \wedge C_{R1}^b, \mathbb{V}\rangle, \langle C_{R2} \uplus H_1', A_1' = A_2' \wedge C_{R2}^b, \mathbb{V}\rangle$. □

If we try to directly transfer the confluence property of a GTS to the corresponding GTS–CHR program, we cannot succeed however, as in general there are too many critical CHR pairs that could cause the GTS–CHR program to become non-confluent. The following example provides a rule which only has one critical GTS pair, but for which the corresponding CHR rule has three critical CHR pairs.

*Example 4.1*

Consider the graph production rule in Figure 10. It removes a loop from a node and has the following corresponding CHR rule:

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

To investigate confluence one must overlap this rule with itself which yields the following three CHR overlap states:

(1) $\langle \mathrm{node}(N, D) \uplus \mathrm{edge}(E, N, N) \uplus \mathrm{edge}(E', N', N'), N = N', \{N, D, E, E', N'\}\rangle$
(2) $\langle \mathrm{node}(N, D) \uplus \mathrm{node}(N', D') \uplus \mathrm{edge}(E, N, N), N = N', \{N, D, N', D', E\}\rangle$
(3) $\langle \mathrm{node}(N, D) \uplus \mathrm{edge}(E, N, N), \top, \{N, D, E\}\rangle$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

State (4.1) is not critical, because the corresponding pair of graph transformations is parallel independent (cf. Ehrig *et al.* 2006), and hence, directly joinable by applying the rule again. State (4.1) is an invalid state, i.e. it violates $\mathscr{G}$, as it has multiple encodings of the same node and state (4.1) is the encoding of the corresponding critical pair for the graph production rule.

As we want to rule out invalid states, we use the following notion of observable confluence presented in Duck *et al.* (2007). It is based on restricting confluence investigations to states that satisfy an invariant. Based on these invariants, observable confluence (or $\mathscr{I}$-confluence) is defined as follows:

*Definition 4.5* (*Observable Confluence*)
A CHR program $P$ is $\mathscr{I}$-confluent with respect to invariant $\mathscr{I}$ if the following holds for all states $\sigma, \sigma_1$ and $\sigma_2$, where $\mathscr{I}(\sigma)$ holds: If $\sigma_1 \ {}^{*}\!\leftarrowtail \sigma \rightarrowtail^{*} \sigma_2$ then $\sigma_1$ and $\sigma_2$ are joinable.

In order to use the graph invariant $\mathscr{G}$ for the notion of observable confluence, we have to investigate the properties of this invariant. We introduce the following definitions from Duck *et al.* (2007). As overlap states themselves may not satisfy the invariant we have to examine all possible extensions that satisfy it. Note that in Duck *et al.* (2007) CHR states are defined as 5-tuples consisting of a goal, user store, built-in store, token store and the set of global variables. As such a verbose definition is not necessary for the remainder of this work, we use the more concise state definition from Section 2.2 and have adjusted the work from Duck *et al.* (2007) accordingly.

*Definition 4.6* (*Extension, Valid Extension*)
A state $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ can be *extended* by another state $\sigma_e = \langle \mathbb{G}_e, \mathbb{B}_e, \mathbb{V}_e \rangle$ as follows.

$$\sigma \lhd \sigma_e = \langle \mathbb{G} \uplus \mathbb{G}_e, \mathbb{B} \wedge \mathbb{B}_e, \mathbb{V}_e \rangle$$

We say that $\sigma_e$ is an *extension* of $\sigma$. A *valid extension* $\sigma_e$ of a state $\sigma$ is an extension such that

$$v \in \mathrm{vars}(\mathbb{G}, \mathbb{B}) \wedge v \notin \mathbb{V} \Rightarrow v \notin \mathrm{vars}(\mathbb{G}_e, \mathbb{B}_e, \mathbb{V}_e).$$

When applied to confluence checking with critical pairs there are generally infinitely many possible extensions of a critical pair. To get a decidable criterion, the following relation on extensions[1] allows us to consider only minimal elements.

*Definition 4.7* (*Relation on Extensions*)
Let $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ be a state, and let $\sigma_{e1} = \langle \mathbb{G}_{e1}, \mathbb{B}_{e1}, \mathbb{V}_{e1} \rangle$ and $\sigma_{e2} = \langle \mathbb{G}_{e2}, \mathbb{B}_{e2}, \mathbb{V}_{e2} \rangle$ be valid extensions of $\sigma$. Then we define $\sigma_{e1} \preceq_\sigma \sigma_{e2}$ to hold if

(1) there exists a valid extension $\sigma_{e3}$ of $(\sigma \lhd \sigma_{e1})$ such that $(\sigma \lhd \sigma_{e1}) \lhd \sigma_{e3} \equiv \sigma \lhd \sigma_{e2}$
(2) $\mathbb{V} - \mathbb{V}_{e1} \subseteq \mathbb{V} - \mathbb{V}_{e2}$ holds.

[1] Originally, in Duck *et al.* (2007) this relation is defined as a partial order, despite being neither transitive nor anti-symmetric. However, it is sufficient for this work to consider it as a reflexive binary relation.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

Note that for any extension $\sigma_e = \langle \mathbb{G}_e, \mathbb{B}_e, \mathbb{V}_e \rangle$ of a state $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ there exists a valid extension $\sigma_\emptyset = \langle \emptyset, \top, \mathbb{V} \rangle$ with $\sigma_\emptyset \preceq_\sigma \sigma_e$, simply because the second condition in Definition 4.7 is trivially satisfied and $\sigma_{e3} = \langle \mathbb{G}_e, \mathbb{B}_e, \mathbb{V}_e \rangle$ satisfies the first condition.

In the following we want to discuss overlap states that do not satisfy an invariant $\mathscr{I}$. Therefore, we are interested in extensions of those states, such that the result satisfies the invariant $\mathscr{I}$. The following definition introduces the set of all those extensions and their minimal elements with respect to the previously defined relation.

*Definition 4.8*
Let $\Sigma_e(\sigma)$ be the set of all valid extensions of a state $\sigma$, and let $\Sigma_e^{\mathscr{I}}(\sigma) = \{\sigma_e \mid \sigma_e \in \Sigma_e(\sigma) \wedge \mathscr{I}(\sigma \lhd \sigma_e)\}$ be the set of all valid extensions satisfying the invariant $\mathscr{I}$. Finally, let $\mathscr{M}_e^{\mathscr{I}}(\sigma)$ be the $\prec_\sigma$-minimal elements of $\Sigma_e^{\mathscr{I}}(\sigma)$.

As shown in Duck *et al.* (2007) the analysis of critical pairs can be extended to this context. Instead of requiring joinability of a critical pair – which might not satisfy the invariant $\mathscr{G}$ – we require joinability for all possible extensions of a critical pair that satisfy $\mathscr{G}$. We make use of the relation on extensions here, such that we only have to investigate minimal extensions. Note that we implicitly consider minimal elements modulo built-in equivalence, e.g. the built-in store $D = 1$ subsumes equivalent stores, like $D = D' + 1 \wedge D' = 0$.

*Definition 4.9*
A program $\mathscr{P}$ is *minimal extension joinable* if for all critical pairs $\mathscr{CP} = (\sigma_1, \sigma_2)$ with overlap $\sigma_{\mathscr{CP}}$, and for all $\sigma_e \in \mathscr{M}_e^{\mathscr{I}}(\sigma_{\mathscr{CP}})$, we have that $(\sigma_1 \lhd \sigma_e, \sigma_2 \lhd \sigma_e)$ is joinable.

It has been shown in Duck *et al.* (2007) that joinability of critical pairs, stemming from overlaps with minimal extensions, is a necessary and sufficient criterion for $\mathscr{I}$-local-confluence if the relation on extensions is well founded.

*Lemma 10* (*Deciding $\mathscr{I}$-Local-Confluence*)
Given that $\prec_{\sigma_{\mathscr{CP}}}$ is well founded for all overlaps $\sigma_{\mathscr{CP}}$, then: $\mathscr{P}$ is $\mathscr{I}$-local-confluent if and only if $\mathscr{P}$ is minimal extension joinable.

Although, in our programs built-in constraints $+$ and $-$ occur, we can consider $\prec_{\sigma_{\mathscr{CP}}}$ well founded for the following reason: On state components other than the built-in store the $\prec_{\sigma_{\mathscr{CP}}}$-relation corresponds to the well-founded subset ordering with the minimal element $\emptyset$ (cf. Duck *et al.* 2007). For the built-ins, we can consider $+$ and $-$ as successor/predecessor terms (as they are only used with constants in rules), and hence, we get well foundedness via proposition 1 of Duck *et al.* (2007).

We further note that for any extension $\sigma_e$ and state $\sigma_{\mathscr{CP}}$ holds that $\sigma_\emptyset \prec_{\sigma_{\mathscr{CP}}} \sigma_e$. The following discussion shows that either $\mathscr{M}_e^{\mathscr{G}}(\sigma_{\mathscr{CP}}) = \{\sigma_\emptyset\}$ or $\Sigma_e^{\mathscr{G}}(\sigma_{\mathscr{CP}}) = \mathscr{M}_e^{\mathscr{G}}(\sigma_{\mathscr{CP}}) = \emptyset$. Whether the minimal element $\sigma_\emptyset$ exists depends solely on $\mathscr{G}(\sigma_{\mathscr{CP}})$ holding as the following lemma shows.

*Lemma 11* (*No Minimal Elements*)
If $\mathscr{G}(\sigma_{\mathscr{CP}})$ is violated for an overlap $\sigma_{\mathscr{CP}}$ then no extension $\sigma_e$ exists such that $\mathscr{G}(\sigma_{\mathscr{CP}} \lhd \sigma_e)$ is satisfied, i.e. $\Sigma_e^{\mathscr{G}}(\sigma_{\mathscr{CP}}) = \mathscr{M}_e^{\mathscr{G}}(\sigma_{\mathscr{CP}}) = \emptyset$.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

*Proof*

We proof this by a structural analysis of the overlap which gives the different possibilities for $\mathscr{G}(\sigma_{\mathscr{CP}})$ to be violated. W.l.o.g. the overlap stems from the two rules $\varrho(r_1) = (r_1 \ @ \ C_{L_1} \Leftrightarrow C^u_{R_1}, C^b_{R_1})$ and $\varrho(r_2) = (r_2 \ @ \ C_{L_2} \Leftrightarrow C^u_{R_2}, C^b_{R_2})$ with the corresponding rule graphs $L_1, L_2, K_1, K_2, R_1$ and $R_2$.

First consider the case of nodes $v_1$ and $v_2$ being overlapped:
Let $\text{type}_{L_1}(v_1)(\text{var}(v_1), D_1) \in C_{L_1}$ and $\text{type}_{L_2}(v_2)(\text{var}(v_2), D_2) \in C_{L_2}$ be overlapped with $\text{type}_{L_1}(v_1) = \text{type}_{L_2}(v_2)$. The equality constraint $\text{var}(v_1) = \text{var}(v_2) \in \sigma_{\mathscr{CP}}$ resembles the merging of the two graph nodes $v_1$ and $v_2$. However, for the degree equalities different possibilities exist:

- $D_1$ and $D_2$ are constants: Then $D_1 = D_2 = \deg_{L_1}(v_1) = \deg_{L_2}(v_2) = k$, as the overlap is impossible otherwise. Then $\sigma_{\mathscr{CP}}$ contains only one constraint $\text{type}_{L_1}(v_1)(\text{var}(v_1), \deg_{L_1}(v_1))$. As in $L_1$ and $L_2$ the nodes each have $k$ adjacent edges, all constraints corresponding to adjacent edges in both rule graphs have to be contained in the overlap as well. If at least one such constraint is not part of the overlap then $\sigma_{\mathscr{CP}}$ contains more than $k$ constraints corresponding to edges adjacent to $v_1 = v_2$. As the degree for the node is a constant it cannot be changed by any extension and the additional edge constraints cannot be removed either. Therefore in such a case, no extension $\sigma_e$ can correct the degree inconsistency and $\mathscr{G}(\sigma_{\mathscr{CP}} \lhd \sigma_e)$ cannot hold.
- $D_1$ and $D_2$ are variable: In this case the overlap is possible without any problems. Depending on the number of overlapped adjacent edge constraints the degree variables can always be instantiated with the correct degree, thus satisfying the invariant $\mathscr{G}$.
- w.l.o.g. $D_1 = k$ and $D_2$ is a variable: this means $D_2 = k \in \sigma_{\mathscr{CP}}$, therefore, all edge constraints of $C_{L_2}$ of edges adjacent to $v_2$ have to be overlapped with edge constraints of $C_{L_1}$ corresponding to edges adjacent to $v_1$. If there is such an edge constraint from $C_{L_2}$ which is not contained in the overlap, then $\sigma_{\mathscr{CP}}$ contains more than $k$ edge constraints corresponding to edges adjacent to $v_1$. Again the degree of $v_1$ is specified as the constant $k$ in $\sigma_{\mathscr{CP}}$, and thus, an extension cannot correct this degree inconsistency. If however, all these edge constraints are contained in the overlap, $\mathscr{G}$ is satisfied again, as there are exactly $k$ such edge constraints coming from $C_{L_1}$.

Finally, consider an edge being overlapped:
Let $\text{type}_{L_1}(\text{var}(e_1), \text{var}(\text{src}(e_1)), \text{var}(\text{tgt}(e_1))) \in C_{L_1}$ and
$\text{type}_{L_2}(\text{var}(e_2), \text{var}(\text{src}(e_2)), \text{var}(\text{tgt}(e_2))) \in C_{L_2}$, then
$\text{var}(e_1) = \text{var}(e_2) \wedge \text{var}(\text{src}(e_1)) = \text{var}(\text{src}(e_2)) \wedge \text{var}(\text{tgt}(e_1)) = \text{var}(\text{tgt}(e_2)) \in \sigma_{\mathscr{CP}}$.
By Definition 3.3 we have constraints $\text{type}_{L_1}(\text{src}(e_1))(\text{var}(\text{src}(e_1)), _) \in C_{L_1}$ and $\text{type}_{L_2}(\text{src}(e_2))(\text{var}(\text{src}(e_2)), _) \in C_{L_2}$. If these two constraints are not part of the overlap, the corresponding equality constraint $\text{var}(\text{src}(e_1)) = \text{var}(\text{src}(e_2)) \in \sigma_{\mathscr{CP}}$ results in a single graph node being represented by two constraints. This is a violation of $\mathscr{G}$, as $\texttt{chr}(\text{ground}, G)$ contains exactly one constraint for each node. This violation cannot be fixed by an extension, as the conflicting additional node

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

constraint cannot be removed. Analogously, the two node constraints corresponding to $\mathrm{tgt}(e_1)$ and $\mathrm{tgt}(e_2)$ have to be contained in the overlap.

Therefore, an overlap $\sigma_{\mathscr{C}\mathscr{P}}$ which violates the invariant $\mathscr{G}$ has to violate it due to one of the above reasons for which it cannot be extended by an extension $\sigma_e$ such that $\mathscr{G}(\sigma_{\mathscr{C}\mathscr{P}} \lhd \sigma_e)$ is satisfied. □

Combining these two results yields the criterion in Corollary 12 for deciding $\mathscr{G}$-local-confluence. Note that this decision criterion is essentially the same criterion as used for traditional local confluence, except that the invariant $\mathscr{G}$ restricts the set of investigated overlaps.

*Corollary 12* (*Deciding $\mathscr{G}$-Local-Confluence*)
$\mathscr{P}$ is $\mathscr{G}$-local-confluent if and only if for all critical pairs $\mathscr{C}\mathscr{P} = (\sigma_1, \sigma_2)$ with overlap $\sigma_{\mathscr{C}\mathscr{P}}$, for which $\mathscr{G}(\sigma_{\mathscr{C}\mathscr{P}})$ holds, $\mathscr{C}\mathscr{P}$ is joinable.

*Proof*
This follows from the combination of Lemma 10, Lemma 11 and the insight that $\sigma_\emptyset$ is the unique minimal extension in the case of $\mathscr{G}(\sigma_{\mathscr{C}\mathscr{P}})$ holding. □

Next we transfer the joinability of critical CHR pairs to strong joinability in GTS:

*Lemma 13* (*$\mathscr{G}$-Confluence Implies Strong Joinability*)
If a terminating GTS–CHR program is $\mathscr{G}$-confluent, then all critical GTS pairs are strongly joinable.

*Proof*
Let $P_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} P_2$ be a critical GTS pair. Let $r_i = (L_i \leftarrow K_i \rightarrow R_i)$ and $\varrho(r_i) = (r_i \mathbin{@} C_{L_i} \Leftrightarrow C^u_{R_i}, C^h_{R_i})$ for $i = 1, 2$.

By Lemma 9 there exists an overlap $\sigma_{\mathscr{C}\mathscr{P}}$ which is a $\mathscr{G}$-state based on $G$. As the critical pair $(\sigma_1, \sigma_2)$ created by the overlap $\sigma_{\mathscr{C}\mathscr{P}}$ is joinable we have the computations $\sigma_{\mathscr{C}\mathscr{P}} \rightarrowtail \sigma_1 \rightarrowtail^* \tau_1$ and $\sigma_{\mathscr{C}\mathscr{P}} \rightarrowtail \sigma_2 \rightarrowtail^* \tau_2$ with $\tau_1 \equiv \tau_2$. From Theorem 5 we know that there exist corresponding GTS transformations $G \overset{r_1,m_1}{\Longrightarrow} P_1 \Longrightarrow^* X_1 \simeq X_2 {}^*\!\Longleftarrow P_2 \overset{r_2,m_2}{\Longleftarrow} G$. The isomorphism between $X_1$ and $X_2$ follows from Lemma 4. Hence, the critical GTS pair is joinable.

To see that it is strongly joinable consider the set $\mathscr{S}(\sigma_{\mathscr{C}\mathscr{P}})$. Every node $v$ for which $\mathrm{tr}_{G \Rightarrow P_1}(v)$ and $\mathrm{tr}_{G \Rightarrow P_2}(v)$ are defined is a node which is not deleted by either $r_1$ or $r_2$. As $m_1$ and $m_2$ are jointly surjective w.l.o.g. there exists a node $v' \in V_{L_1}$ of rule $r_1$ with $m(v') = v$. As the node is not removed we know $v' \in V_{K_1}$, and therefore, $\mathrm{type}_{K_1}(v')(\mathrm{var}(v'), \mathrm{dvar}(v')) \in C_{L_1}$. Either the node is not part of the overlap in $\sigma_{\mathscr{C}\mathscr{P}}$, or if it is overlapped with a node $v'' \in V_{L_2}$ such that $m(v') = m(v'')$, then we also know that $v'' \in V_{K_2}$ due to the defined track morphism. Therefore, we always have the node constraint $\mathrm{type}_{K_1}(v')(\mathrm{var}(v), \mathrm{dvar}(v)) \in \sigma_{\mathscr{C}\mathscr{P}}$ and $v \in \mathscr{S}(\sigma_{\mathscr{C}\mathscr{P}})$. As this node cannot be removed during the transformation, a variant of this constraint with adjusted degree is also present in $\tau_1$ and $\tau_2$. These two variant constraints are uniquely determined, as $\mathrm{var}(v) \in \mathbb{V}$ by Definition 4.4, and hence, they both have to use $\mathrm{var}(v)$ for the node identifier variable. This means we still have to showfor such

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

a node $v$ that the two conditions from Definition 4.2 are satisfied:

(1) $\mathrm{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)$ and $\mathrm{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$ are defined:
By Theorem 5 we know that the GTS transformations are strong w.r.t. $\mathscr{S}(\sigma_{\mathscr{C}\mathscr{P}})$. As $v \in \mathscr{S}(\sigma_{\mathscr{C}\mathscr{P}})$ this implies $v \in m(K) \vee v \notin m(L)$ for each of the applied rules, i.e. the node remains during the transformation and hence the track morphisms are defined as in Definition 2.5.

(2) $f_V(\mathrm{tr}_{G \Rightarrow P_1 \Rightarrow X_1}(v)) = \mathrm{tr}_{G \Rightarrow P_2 \Rightarrow X_2}(v)$:
As the isomorphism $f$ is derived from $\tau_1 \equiv \tau_2$ and $\mathrm{var}(v) \in \mathbb{V}$ this isomorphism correctly relates the original node $v$ with its occurrences in $\tau_1$, resp. $X_1$ and $\tau_2$, resp. $X_2$. □

The reverse direction holds as well, as the following lemma shows.

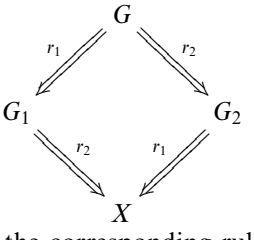
*Lemma 14* (*Strong Joinability Implies $\mathscr{G}$-Confluence*)
If all critical GTS pairs of a terminating GTS are strongly joinable, then the corresponding GTS–CHR program is $\mathscr{G}$-confluent.

*Proof*
Consider an overlap $\sigma_{\mathscr{C}\mathscr{P}}$ for the critical CHR pair $(\sigma_1, \sigma_2)$. W.l.o.g. $\mathscr{G}(\sigma_{\mathscr{C}\mathscr{P}})$ holds according to Corollary 12. Therefore, $\sigma_{\mathscr{C}\mathscr{P}}$ is a $\mathscr{G}$-state based on $G$ and $\sigma_1, \sigma_2$ correspond to graphs $G_1, G_2$. Consider now $G_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} G_2$.

We now show that either the critical CHR pair is non-critically joinable, or it corresponds to a critical GTS pair and can thus be joined, because all critical GTS pairs are strongly joinable.

First, we want to point out that $G$ is minimal by the definition of the CHR overlap, i.e. every occurring node and edge is part of a match, hence, $m_1$ and $m_2$ are jointly surjective.

Next, we distinguish two cases: First, let $G_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} G_2$ be parallel independent. Therefore, the second rule can be applied after the first, because none of the required nodes or edges has been removed. The following diagram depicts this situation:

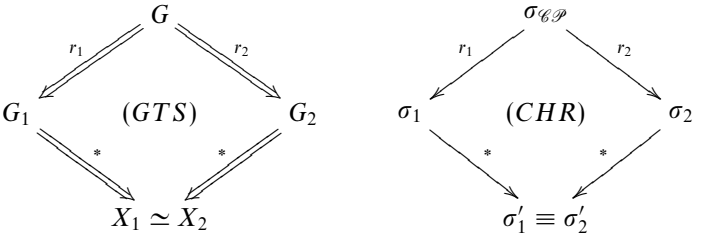



By Theorem 5 we can apply the corresponding rules to $\sigma_{\mathscr{C}\mathscr{P}}$ in order to join the critical CHR pair, because $\mathscr{S}(\sigma_{\mathscr{C}\mathscr{P}})$ contains only nodes not deleted by $r_1$ and $r_2$.

Secondly, let $G_1 \overset{r_1,m_1}{\Longleftarrow} G \overset{r_2,m_2}{\Longrightarrow} G_2$ be parallel dependent. It follows that $m(L_1) \cap m(L_2) \not\subseteq m(K_1) \cap m(K_2)$. However, this is now a critical GTS pair, and hence,

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

strongly joinable as depicted on the left of the following diagram:




The right part of the diagram shows the situation for the critical CHR pair which is joinable by Theorem 5. This is possible, because $\forall v \in \mathscr{S}(\sigma_{\mathscr{C}\mathscr{P}})$ we know that $\mathrm{tr}_{G \Rightarrow G_1}(v)$ and $\mathrm{tr}_{G \Rightarrow G_2}(v)$ are defined, thus by Definition 4.2, $v$ is never removed and still present in $X_1$ and $X_2$. Finally, the isomorphism implied by $X_1 \simeq X_2$ gives us $\sigma'_1 \equiv \sigma'_2$. Note that despite Lemma 4 not being reversible in general this holds here, as it is clearly determined for both $\sigma'_1$ and $\sigma'_2$ which node identifier variables are global and the strong joinability condition reflects this in the isomorphism.

Therefore, for all overlaps $\sigma_{\mathscr{C}\mathscr{P}}$ with $\mathscr{G}(\sigma_{\mathscr{C}\mathscr{P}})$ holding we know that the corresponding critical CHR pair is joinable, and hence, by Corollary 12 that the CHR program is $\mathscr{G}$-local-confluent. As it is terminating as well, it is $\mathscr{G}$-confluent. □

The combination of the previous two lemmata gives us our main result:

*Theorem 15* (*Strong Joinability iff $\mathscr{G}$-Confluence*)
All critical GTS pairs of a terminating GTS are strongly joinable if and only if the corresponding GTS–CHR program is $\mathscr{G}$-confluent.

*Proof*
Direct combination of Lemmas 13 and 14. □

*Corollary 16* (*$\mathscr{G}$-Confluence Implies GTS Confluence*)
If a terminating GTS–CHR program is $\mathscr{G}$-confluent, then the corresponding GTS is confluent.

*Proof*
Strong joinability is a sufficient criterion for confluence of a terminating GTS (cf. Plump 2005). Therefore, this follows directly from Theorem 15. □

Practically, with Theorem 15 we can reuse the automatic confluence check for terminating CHR programs (Abdennadher *et al*. 1999; Frühwirth 2009) to prove confluence of a terminating GTS–CHR program. As Lemma 11 showed, it is sufficient to only consider overlaps satisfying the graph invariant $\mathscr{G}$. Whenever all the resulting critical CHR pairs are joinable, the CHR program is $\mathscr{G}$-confluent according to Corollary 12. This, in turn, is sufficient for proving confluence of the original GTS.

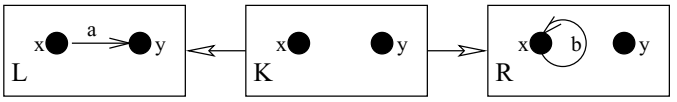
### 4.3 Discussion

In this section we elaborate on some canonical examples that highlight different properties of critical pairs. These examples are inspired by Plump (2005).

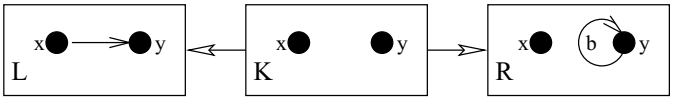
https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

*Example 4.2*
Consider the following rules which use two different edge types: (a) and (b)

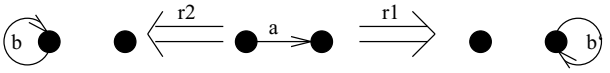
r1:



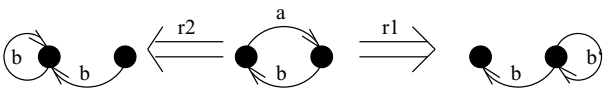

r2:




The only critical GTS pair of these rules is joinable. This is possible in the GTS case, because the resulting graphs, shown below, are isomorphic.




However, the track morphisms of the above derivations are incompatible, i.e. the strong joinability condition from Definition 4.2 cannot be satisfied. As the following derivation shows, this hinders monotonicity and joinability is lost, when the critical pair is embedded into a larger context.




The two resulting states are no longer isomorphic and also cannot be joined, as no more rules are applicable to them. Therefore, this GTS is not locally confluent, although all its critical GTS pairs are joinable.

We now examine this scenario in CHR. The two GTS rules then become the following CHR rules:

$$
\begin{aligned}
r1 \ @ \quad & \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{a}(E, N_x, N_y) \\
& \Leftrightarrow \\
& \mathrm{node}(N_x, D'_x) \uplus \mathrm{node}(N_y, D'_y) \uplus \mathrm{b}(E', N_x, N_x), \\
& D'_x = D_x{+}1 \wedge D'_y = D_y{-}1 \\
r2 \ @ \quad & \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{a}(E, N_x, N_y) \\
& \Leftrightarrow \\
& \mathrm{node}(N_x, D'_x) \uplus \mathrm{node}(N_y, D'_y) \uplus \mathrm{b}(E', N_y, N_y), \\
& D'_x = D_x{-}1 \wedge D'_y = D_y{+}1
\end{aligned}
$$

We now consider the critical CHR pair corresponding to the above critical GTS pair. It is generated by fully overlapping both rule heads, resulting in the overlap

$$\sigma_{\mathscr{C}\mathscr{P}} = \langle \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N_2, D_2) \uplus \mathrm{a}(E, N_1, N_2), \top, \mathbb{V} \rangle$$

with $\mathbb{V} = \{N_1, N_2, D_1, D_2, E\}$. The resulting critical CHR pair $(\sigma_1, \sigma_2)$ is:

$$
\begin{aligned}
& \langle \mathrm{node}(N_1, D'_1) \uplus \mathrm{node}(N_2, D'_2) \uplus b(E', N_1, N_1), D'_1 = D_1{+}1 \wedge D'_2 = D_2{-}1, \mathbb{V} \rangle, \\
& \langle \mathrm{node}(N_1, \tilde{D}_1) \uplus \mathrm{node}(N_2, \tilde{D}_2) \uplus b(\tilde{E}, N_2, N_2), \tilde{D}_1 = D_1{-}1 \wedge \tilde{D}_2 = D_2{+}1, \mathbb{V} \rangle
\end{aligned}
$$

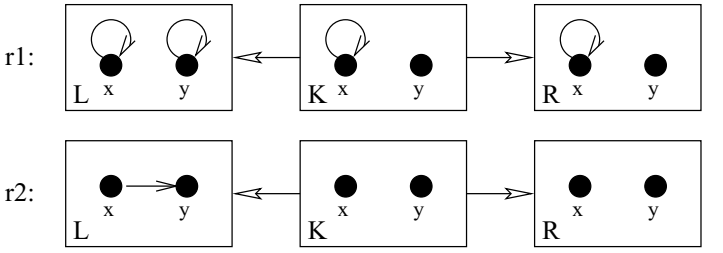
https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

It is clear that $\sigma_1 \not\equiv \sigma_2$, because $\mathscr{CT} \not\models (D_1' = D_1{+}1 \wedge D_2' = D_2{-}1) \rightarrow \exists_\emptyset N_1 = N_2$ as required by Theorem 2.

The strong nodes $N_1$ and $N_2$, i.e. $N_1, N_2 \in \mathbb{V}$, enforce compatible track morphisms, and hence are responsible for the non-joinability above. If we instead want to test non-strong joinability, we can do so as well by setting $\mathbb{V} = \emptyset$. Then, the two states $\sigma_1$ and $\sigma_2$ are indeed equivalent by Definition 2.8, as $N_2$ is existentially quantified and the remaining conditions of Theorem 2 hold as well.

*Example 4.3*

Another example from Plump (2005) is the following GTS which is terminating and confluent; however, the critical GTS pair from the overlap of rule $r1$ with itself is not strongly joinable. This is a counterexample used to show that strong joinability of critical GTS pairs is only a sufficient criterion for confluence of a terminating GTS.




The GTS works as follows: If there is at least one loop in the graph, then all but a last loop are removed by the first rule. Additionally, all non-loop edges are removed by the second rule. Therefore, the remaining final graph contains zero or one loops and no other edges, and hence the GTS is terminating and confluent due to graph isomorphism. The first rule is encoded in CHR as follows:

$$\begin{aligned} r1 \ @ \quad & \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \\ & \mathrm{a}(E_x, N_x, N_x) \uplus \mathrm{a}(E_y, N_y, N_y) \\ & \Leftrightarrow \\ & \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y') \uplus \mathrm{a}(E_x, N_x, N_x), \\ & D_y' = D_y - 2 \end{aligned}$$

Completely overlapping the rule with itself yields the overlap

$$\sigma_{\mathscr{CP}} = \langle \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N_2, D_2) \uplus \mathrm{a}(E_1, N_1, N_1) \uplus \mathrm{a}(E_2, N_2, N_2), \top, \mathbb{V} \rangle$$

with $\mathbb{V} = \{N_1, N_2, D_1, D_2, E_1, E_2\}$ resulting in the critical CHR pair $(\sigma_1, \sigma_2)$ with:

$$\begin{aligned} \sigma_1 &= \langle \mathrm{node}(N_1, D_1) \uplus \mathrm{node}(N_2, D_2') \uplus \mathrm{a}(E_1, N_1, N_1), D_2' = D_2 - 2, \mathbb{V} \rangle \\ \sigma_2 &= \langle \mathrm{node}(N_1, D_1') \uplus \mathrm{node}(N_2, D_2) \uplus \mathrm{a}(E_2, N_2, N_2), D_1' = D_1 - 2, \mathbb{V} \rangle \end{aligned}$$

Analogously to the previous example, the two states are not equivalent and cannot be joined, therefore the corresponding critical GTS pair is not strongly joinable. Again, setting $\mathbb{V} = \emptyset$ results in both states becoming equivalent. As before, this reflects that for the critical GTS pairs the two corresponding graphs are isomorphic.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

## 5 Analysing operational equivalence

Constraint Handling Rules is well known for its decidable, sufficient, and necessary criterion for operational equivalence of terminating and confluent programs (Abdennadher and Frühwirth 1999; Frühwirth 2009). After presenting this result in Section 5.1, we introduce the concept of operational equivalence for GTS in Section 5.2. Then we investigate operational equivalence of GTS–CHR programs and show that it is sufficient for operational equivalence of the original GTS. We further demonstrate its application to detect redundant rules of a GTS.

The contents of this section are a revised and extended version of (Raiser and Frühwirth 2009a).

### *5.1 Operational equivalence in CHR*

Operational equivalence, intuitively, means that two programs should be able to compute equivalent outputs given the same input. Applied to a single state, this behaviour is called $\mathscr{P}_1, \mathscr{P}_2$-joinability:

*Definition 5.1* ($\mathscr{P}_1, \mathscr{P}_2$*-joinability*)
Let $\mathscr{P}_1, \mathscr{P}_2$ be CHR programs. A state $\sigma$ is $\mathscr{P}_1, \mathscr{P}_2$*-joinable*, if and only if there are computations $\sigma \rightarrowtail^*_{\mathscr{P}_1} \sigma_1$ and $\sigma \rightarrowtail^*_{\mathscr{P}_2} \sigma_2$ with $\sigma_1 \equiv \sigma_2$, where all $\sigma_i$ are final states with respect to $\mathscr{P}_i$.

If $\mathscr{P}_1, \mathscr{P}_2$-joinability is given for all states the programs are considered operationally equivalent:

*Definition 5.2* (*Operational Equivalence*)
Let $\mathscr{P}_1, \mathscr{P}_2$ be CHR programs.

$\mathscr{P}_1, \mathscr{P}_2$ are *operationally equivalent* if and only if all states $\sigma$ are $\mathscr{P}_1, \mathscr{P}_2$-joinable.

As mentioned before, operational equivalence is decidable for terminating and confluent CHR programs. Similarly to confluence, the decision algorithm investigates critical states created from rule heads.

*Definition 5.3* (*Critical States*)
Let $\mathscr{P}_1, \mathscr{P}_2$ be CHR programs. The set of *critical states of* $\mathscr{P}_1$ *and* $\mathscr{P}_2$ is defined as $\{\langle H, \top, \mathrm{vars}(H)\rangle \mid (H \Leftrightarrow B_c, B_b) \in \mathscr{P}_1 \cup \mathscr{P}_2\}$.

Note that we had to consider observable confluence for CHR, because overlap states constructed for critical pair analysis may not always encode a graph. The critical states used for operational equivalence here, however, stem directly from a complete head of a rule, which in turn was derived from a GTS rule graph. Therefore, all critical states of GTS–CHR programs are valid encodings of graphs.

The following theorem, adapted from Abdennadher and Frühwirth (1999), is based on the idea to determine $\mathscr{P}_1, \mathscr{P}_2$-joinability of these critical states. The monotonicity property of CHR ensures that if all critical states are $\mathscr{P}_1, \mathscr{P}_2$-joinable, then all states are. Additionally demanding termination and confluence of the programs, allows

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

us to decide $\mathscr{P}_1, \mathscr{P}_2$-joinability simply by executing a critical state in each of the programs and then comparing the resulting final states.

*Theorem 17* (*Operational Equivalence via Critical States*)
Let $\mathscr{P}_1, \mathscr{P}_2$ be terminating and confluent CHR programs. $\mathscr{P}_1, \mathscr{P}_2$ are operationally equivalent if and only if for all critical states $\sigma$ of $\mathscr{P}_1$ and $\mathscr{P}_2$ it holds that $\sigma$ is $\mathscr{P}_1, \mathscr{P}_2$-joinable.

*Proof*
Given in Abdennadher and Frühwirth (1999). □

Note that in contrast to confluence, Theorem 17 will always consider states satisfying the $\mathscr{G}$-invariant, when applied to a GTS–CHR program. This follows from the fact that each critical state is the head of a rule, and in turn, corresponds to a rule graph from the GTS by construction.

### 5.2 Analysing operational equivalence in GTS

In this section we introduce the notion of operational equivalence for GTS. Based on the previous embedding of GTS in CHR, we use the existing decision algorithm from CHR as a sufficient criterion for operational equivalence of two GTS.

First, we define the property of $\mathscr{S}_1, \mathscr{S}_2$-joinability for two GTS $\mathscr{S}_1, \mathscr{S}_2$, analogously to $\mathscr{P}_1, \mathscr{P}_2$-joinability.

*Definition 5.4* ($\mathscr{S}_1, \mathscr{S}_2$-*joinability*)
Let $\mathscr{S}_1, \mathscr{S}_2$ be two GTS. A typed graph $G$ is $\mathscr{S}_1, \mathscr{S}_2$-*joinable* if and only if there are derivations $G \Rightarrow^*_{\mathscr{S}_1} G_1$ and $G \Rightarrow^*_{\mathscr{S}_2} G_2$ with $G_1 \simeq G_2$ being final with respect to $\mathscr{S}_1$ and $\mathscr{S}_2$.

Here $\simeq$ denotes traditional graph isomorphism and a graph $G$ is considered *final* with respect to $\mathscr{S}$ iff there is no transition $G \Rightarrow_{\mathscr{S}} H$ for any graph $H$.

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

*Definition 5.5* (*GTS Operational Equivalence*)
Let $\mathscr{S}_1 = (\mathscr{P}_1, TG)$ and $\mathscr{S}_2 = (\mathscr{P}_2, TG)$ be two GTS.

$\mathscr{S}_1, \mathscr{S}_2$ are *operationally equivalent* if and only if for all graphs $G$ typed over $TG$ it holds that $G$ is $\mathscr{S}_1, \mathscr{S}_2$-joinable.

Similar to operational equivalence in CHR, where it is futile to directly compare programs that use different constraints, Definition 5.5 requires $\mathscr{S}_1$ and $\mathscr{S}_2$ to be based on the same type graph $TG$. With the previous results from Raiser and Frühwirth (2009b) we can directly use CHR's operational equivalence as a sufficient criterion for deciding operational equivalence of two GTS:

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

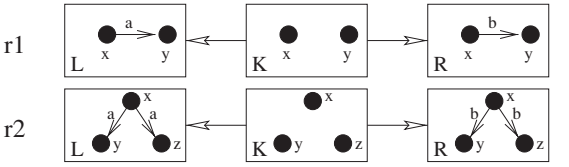



Fig. 11. Example of a graph transformation system.

*Theorem 18* (*GTS–CHR Operational Equivalence*)

Let $\mathscr{S}_1, \mathscr{S}_2$ be GTS and $\mathscr{P}_1, \mathscr{P}_2$ their corresponding GTS–CHR programs. $\mathscr{S}_1, \mathscr{S}_2$ are operationally equivalent if $\mathscr{P}_1, \mathscr{P}_2$ are operationally equivalent.

*Proof*

Let $G$ be a graph typed over $TG$. Then the state $\sigma = \langle \mathtt{chr}(\mathrm{ground}, G), \top, \emptyset \rangle$ is $\mathscr{P}_1, \mathscr{P}_2$-joinable by Definition 5.1. Therefore, there exist the final states $\sigma_1 \equiv \sigma_2$ with $\sigma \rightarrowtail^*_{\mathscr{P}_1} \sigma_1$ and $\sigma \rightarrowtail^*_{\mathscr{P}_2} \sigma_2$.

By Theorem 5 we know that there exist corresponding derivations $G \Rightarrow^*_{\mathscr{S}_1} G_1$ and $G \Rightarrow^*_{\mathscr{S}_2} G_2$ such that $\sigma_1$ is a $\mathscr{G}$-state based on $G_1$ and $\sigma_2$ is a $\mathscr{G}$-state based on $G_2$.

The graphs $G_1$ and $G_2$ are final states w.r.t. $\mathscr{S}_1$ and $\mathscr{S}_2$, and finally, the isomorphism between $G_1$ and $G_2$ is implied by $\sigma_1 \equiv \sigma_2$ according to Lemma 4. Therefore, $G$ is $\mathscr{S}_1, \mathscr{S}_2$-joinable. □

An interesting application of the above theorem is the removal of redundant rules. Originally proposed in Abdennadher and Frühwirth (1999), decidable operational equivalence of CHR programs implies a straightforward redundant rule removal algorithm: Remove a single rule from the program, then compare the operational equivalence of the program thus created and the original program. If the two programs are operationally equivalent the selected rule is shown to be redundant and can be removed.

Clearly, program equivalence in general is undecidable, and hence, we cannot expect such an algorithm to correctly identify all redundant rules. Nevertheless, the algorithm was applied in CHR research to great success on automatically generated programs (Abdennadher and Sobhi 2007; Raiser 2008). These generations tend to create rules which subsume each other, in which case the algorithm works well as the following example demonstrates.

*Example 5.1*

Consider the graph transformation system $\mathscr{S}_1$ given in Figure 11. It depicts a typical case, in that the rule $r2$ is subsumed by rule $r1$. While this is easily verified by a human reader, Theorem 17 gives us the means for an automated verification.

In order to verify the redundancy of rule $r2$, consider a second graph transformation system $\mathscr{S}_2$, which contains only rule $r1$. Proving that $\mathscr{S}_1$ and $\mathscr{S}_2$ are operationally equivalent then proves the redundancy of rule $r2$.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

Encoding the graph transformation system $\mathscr{S}_1$ from Figure 11 in CHR results in the following two rules:

$$r1 \ @ \quad \begin{array}{l} \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{a}(E, N_x, N_y) \\ \Leftrightarrow \\ \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{b}(E', N_x, N_y) \end{array}$$

$$r2 \ @ \quad \begin{array}{l} \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{node}(N_z, D_z) \uplus \\ \mathrm{a}(E_y, N_x, N_y) \uplus \mathrm{a}(E_z, N_x, N_z) \\ \Leftrightarrow \\ \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{node}(N_z, D_z) \uplus \\ \mathrm{b}(E_1, N_x, N_y) \uplus \mathrm{b}(E_2, N_x, N_z) \end{array}$$

This GTS–CHR program $\mathscr{P}_1$ is confluent and terminating and the same holds for $\mathscr{P}_2$, which encodes $\mathscr{S}_2$ respectively. Next, we investigate the $\mathscr{P}_1, \mathscr{P}_2$-joinability of all critical states, of which there are two. The critical state derived from $r1$ is clearly $\mathscr{P}_1, \mathscr{P}_2$-joinable, as the same rule can be applied to it in both programs, resulting in equivalent final states.

The critical state derived from rule $r2$ contains two a-edges, which can be converted to b-edges either by applying rule $r2$ or rule $r1$ twice. Therefore, the final states in both programs are equivalent again, and hence, the programs are operationally equivalent. As a conclusion, $\mathscr{S}_1$ and $\mathscr{S}_2$ are operationally equivalent, which in turn proves the redundancy of rule $r2$.

In general, Theorem 18 cannot be reversed, i.e. it is only a sufficient, not a necessary criterion. A counterexample for the reverse direction is given in the following example. Notice that it is based on the example used by Plump (2005) in order to demonstrate why the critical pair lemma is not a necessary criterion for confluence. This might be seen as an indication that a similar situation exists for GTS program equivalence.

*Example 5.2*

Consider two GTS with the first being the one from Example 4.3 and the second GTS is identical to the first except for rule $r1$, in which the loop for node $x$ is removed instead. It is clear that both programs are terminating, confluent and operationally equivalent. The following two rules are from the corresponding GTS–CHR programs $\mathscr{P}_1$ and $\mathscr{P}_2$:

$$r1 \ @ \quad \begin{array}{l} \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \\ \mathrm{a}(E_x, N_x, N_x) \uplus \mathrm{a}(E_y, N_y, N_y) \\ \Leftrightarrow \\ \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D'_y) \uplus \mathrm{a}(E_x, N_x, N_x), \\ D'_y = D_y - 2 \end{array}$$

$$r1' \ @ \quad \begin{array}{l} \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \\ \mathrm{a}(E_x, N_x, N_x) \uplus \mathrm{a}(E_y, N_y, N_y) \\ \Leftrightarrow \\ \mathrm{node}(N_x, D'_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{a}(E_y, N_y, N_y), \\ D'_x = D_x - 2 \end{array}$$

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

We can now investigate the following critical state $\sigma$ according to Theorem 18, where $\mathbb{V} = \{N_x, N_y, D_x, D_y, E_x, E_y\}$:

$$\sigma = \langle \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{a}(E_x, N_x, N_x) \uplus \mathrm{a}(E_y, N_y, N_y), \top, \mathbb{V} \rangle$$

The critical state $\sigma$ is not $\mathscr{P}_1, \mathscr{P}_2$-joinable, as there is only one rule applicable in each program and the resulting states are not equivalent:

$$\sigma \rightarrowtail^{r1}_{\mathscr{P}_1} \langle \mathrm{node}(N_x, D_x) \uplus \mathrm{node}(N_y, D'_y) \uplus \mathrm{a}(E_x, N_x, N_x), D'_y = D_y - 2, \mathbb{V} \rangle = \tau_1$$
$$\not\equiv$$
$$\sigma \rightarrowtail^{r1'}_{\mathscr{P}_2} \langle \mathrm{node}(N_x, D'_x) \uplus \mathrm{node}(N_y, D_y) \uplus \mathrm{a}(E_y, N_y, N_y), D'_x = D_x - 2, \mathbb{V} \rangle = \tau_2$$

## 6 Related and future work

The relation of CHR to other formalisms has been thoroughly investigated in the literature. This includes comparison to logical formalisms (e.g. linear logic Betz and Frühwirth 2005), term rewriting (Duck *et al*. 2006), Join-Calculus (Lam and Sulzmann 2008) and Petri nets (Betz 2007). More detailled surveys of these relations can be found in Sneyers *et al.* (2010) and Frühwirth (2009).

The relation of GTS to CHR differs from these other formalisms, because firstly, it is a graph-based formalism, and secondly, there are significant differences in program analysis results. Most importantly, confluence of terminating GTS is undecidable (Plump 2005), whereas confluence of terminating CHR programs is decidable (Abdennadher *et al*. 1999). Furthermore, no operational equivalence analysis exists for GTS, as opposed to the situation in CHR (Abdennadher and Frühwirth 1999).

The operational equivalence test presented in Section 5 yields a method for removal of redundant rules, which is remarkable for another reason: In Kreowski and Valiente (2000) the notions of redundancy and subsumptions have been introduced for GTS, however, the authors only gave the definitions and a sufficient condition for redundancy, but no verification procedure. While the notion of redundancy in that paper is slightly different from the one found in Abdennadher and Frühwirth (2003), the adaptation of the algorithm to GTS–CHR programs is to the best of our knowledge the only available verification procedure for redundant GTS rules.

Note that operational equivalence, as defined here, is only one possible notion of equivalence between programs. It was used in this work as an example of CHR program analyses applied to embedded GTS. Another, more popular, notion of equivalence of GTS is bisimilarity, introduced in the GTS context in Ehrig and König (2004). It has been successfully applied to determine behavioural equivalence of GTS in Rangel *et al.* (2008). While bisimilarity originated from process calculi and is focused on the transitions made during computations of a result, operational equivalence on the other hand, only compares the final computational results, independently of how they are reached.

The encoding of GTS in CHR, as presented in Section 3, is based on the double-pushout approach for GTS. A related graph rewriting mechanism, the single-pushout approach, was introduced in Löwe (1993). Instead of demanding two pushouts, as

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

in Figure 4, rewriting is defined there over a category of partial graph morphisms, hence only a single pushout construction is used. Intuitively, this results in a different behaviour with respect to dangling edges: While the double-pushout approach prohibits a rule application in case a dangling edge would remain, the single-pushout approach removes all dangling edges instead. In Löwe and Müller (1993) the authors investigate confluence for single-pushout graph rewriting. In particular, the critical pair analysis is shown to be only a sufficient criterion as well, not a necessary one.

In this work, we based our encoding on the DPO approach as the non-applicability of rules due to the dangling edge condition corresponds nicely to non-applicability of corresponding CHR rules. In order to support the approach from Löwe (1993), remaining dangling edges would need to be removed by an additional rule, hence, we would lose the one-on-one correspondence of GTS and CHR rules.

Our encoding further serves as the foundation of the extensible platform for the analysis of GTS using CHR presented in the diploma thesis (Wasserthal 2009). This platform is based on JCHR (Van Weert 2008), a Java-based implementation of CHR and the work presented in Section 3. The developed tool presents a graphical view of a GTS which is synchronized with the corresponding GTS–CHR program at all times. Furthermore, it provides an interface for program analysis plug-ins, which can work directly on the GTS or on the GTS–CHR program.

As this work demonstrated, our embedding leads to cross-fertilizations of CHR and GTS research. Future work should therefore concentrate on further comparing the different approaches to program analysis. In particular, CHR provides several approaches to termination analysis (Frühwirth 2000; Voets *et al*. 2008; Pilozzi and De Schreye 2008) that GTS research may profit from.

Research on GTS contains several extensions for the typed graphs and rules considered in this work. One such extension adds attributes (Ehrig *et al*. 2006) to graphs, which can then be modified by rules. We assume that built-in constraints available in CHR could closely correspond to attributes. Another important extension, is the addition of negative application conditions (Ehrig *et al*. 2006), i.e. applying a rule requires the absence of certain graph structures. This is more difficult to achieve in CHR, as it traditionally has no support for negation as absence. However, there exist proposed extensions of CHR with negation as absence (Van Weert *et al*. 2006) and aggregates (Van Weert *et al*. 2008), which could help in extending our encoding to allow application conditions.

Our work on operational equivalence for GTS yielded a first useable criterion. However, there is lot of remaining work in this field. From a decidability point of view, operational equivalence is in a similar situation as confluence: Plump (2005) showed that confluence is undecidable even for terminating GTS and we expect a similar result for operational equivalence. Therefore, our criterion might only be applicable to a small subset of all GTS.

Similarly, in CHR research, the operational equivalence result assumes that both programs use the same constraint symbols in the same manner. While this restriction yields a decidable criterion, it also means that it seldomly applies to real-world programs. Traditionally, one may be able to manually show operational equivalence

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

for two concrete programs by taking into account known restrictions on data structures or inputs and ignoring irrelevant states. The same situation was present for confluence (e.g. Frühwirth 2005) until observable confluence (Duck *et al*. 2007) succeeded in providing an extended approach.

We plan to develop such an invariant-based approach for operational equivalence in CHR as well, which extends Theorem 17. Combined with a better criterion for operational equivalence in GTS, including the track morphism similarly to the critical pair approach, this might reveal a closer correspondence between operational equivalence in both systems.

## 7 Conclusion

We have shown thatCHR provides an elegant way for embeddingGTS. The resulting rules are concise and directly related to the corresponding graph production rules. We proved soundness and completeness of this embedding and verified formal properties of CHR states that encode graphs. Furthermore, we considered partial graphs and showed that the CHR embedding naturally supports these, hence facilitating program analysis.

Next, we analysed confluence and showed that observable confluence of a GTS–CHR program is a sufficient criterion for confluence of the analysed GTS. Furthermore, we transferred the notion of operational equivalence from CHR to GTS and discussed the CHR-based decision algorithm for redundant rule removal.

## References

Abdennadher, S. and Frühwirth, T. 1999. Operational equivalence of CHR programs and constraints. In *Principles and Practice of Constraint Programming, CP 1999*, J. Jaffar, Ed. Lecture Notes in Computer Science, vol. 1713. Springer-Verlag, 43–57.

Abdennadher, S. and Frühwirth, T. 2003. Integration and optimization of rule-based constraint solvers. In *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25–27, 2003, Revised Selected Papers*, M. Bruynooghe, Ed. Lecture Notes in Computer Science, vol. 3018. Springer-Verlag, 198–213.

Abdennadher, S., Frühwirth, T. and Meuss, H. 1999. Confluence and semantics of constraint simplification rules. *Constraints 4,* 2, 133–165.

Abdennadher, S. and Marte, M. 2000. University course timetabling using Constraint Handling Rules. In *Special Issue on Constraint Handling Rules*, C. Holzbaur and T. Frühwirth, Eds. Journal of Applied Artificial Intelligence, vol. 14(4). Taylor & Francis, 311–325.

Abdennadher, S. and Sobhi, I. 2007. Generation of rule-based constraint solvers: Combined approach. In *Logic-Based Program Synthesis and Transformation, 17th International Symposium, LOPSTR 2007, Kongens Lyngby, Denmark, August 23-24, 2007, Revised Selected Papers*, A. King, Ed. Lecture Notes in Computer Science, vol. 4915. Springer-Verlag, 106–120.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press, New York.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

Bakewell, A., Plump, D. and Runciman, C. 2003. Specifying pointer structures by graph reduction. In *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Revised Selected and Invited Papers*, J. L. Pfaltz, M. Nagl and B. Böhlen, Eds. Lecture Notes in Computer Science, vol. 3062. Springer-Verlag, Charlottesville, VA, 30–44.

Barranco-Mendoza, A. 2005. *Stochastic and Heuristic Modelling for Analysis of the Growth of Pre-Invasive Lesions and for a Multidisciplinary Approach to Early Cancer Diagnosis*. PhD thesis, Simon Fraser University, Burnaby, Canada.

Bavarian, M. and Dahl, V. 2006. Constraint based methods for biological sequence analysis. *Journal of Universal Computer Science 12,* 11, 1500–1520.

Betz, H. 2007. Relating coloured Petri nets to Constraint Handling Rules. In *CHR '07* (Porto, Portugal), K. Djelloul, G. J. Duck and M. Sulzmann, Eds. 33–47.

Betz, H. and Frühwirth, T. 2005. A Linear-Logic Semantics for Constraint Handling Rules. In *Principles and Practice of Constraint Programming, 11th International Conference, CP 2005*, P. van Beek, Ed. Lecture Notes in Computer Science, vol. 3709. Springer-Verlag, Sitges, Spain, 137–151.

Blostein, D., Fahmy, H., and Grbavec, A. 1995. Practical use of graph rewriting. In *Fifth Workshop on Graph Grammars and Their Application To Computer Science*, J. E. Cuny, H. Ehrig, G. Engels and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 1073. Springer-Verlag, 38–55.

Dahl, V. and Maharshak, E. 2009. DNA replication as a model for computational linguistics. In *IWINAC '09: Proceedings of the Third International Work-Conference on the Interplay Between Natural and Artificial Computation*, J. M. Mira, J. M. Ferrández, J. R. Álvarez, F. de la Paz and F. J. Toledo, Eds. Lecture Notes in Computer Science, vol. 5601. Springer-Verlag, 346–355.

Duck, G. J., Stuckey, P. J., and Brand, S. 2006. ACD term rewriting. In *ICLP '06* (Seattle, Washington), S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer-Verlag, 117–131.

Duck, G. J., Stuckey, P. J., and Sulzmann, M. 2007. Observable confluence for constraint handling rules. In *Logic Programming, 23rd International Conference, ICLP 2007*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer-Verlag, Porto, Portugal, 224–239.

Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag.

Ehrig, H. and König, B. 2004. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, I. Walukiewicz, Ed. Lecture Notes in Computer Science, vol. 2987. Springer-Verlag, Barcelona, Spain, 151–166.

Frühwirth, T. 2000. Proving termination of constraint solver programs. In *Selected Papers from the Joint ERCIM/Compulog Net Workshop on New Trends in Constraints*, K. R. Apt, A. C. Kakas, E. Monfroy and F. Rossi, Eds. Lecture Notes in Computer Science, vol. 1865. Springer-Verlag, 298–317.

Frühwirth, T. 2005. Parallelizing union-find in constraint handling rules using confluence analysis. In *Logic Programming, 21st International Conference, ICLP 2005*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer-Verlag, 113–127.

Frühwirth, T. 2009. *Constraint Handling Rules*. Cambridge University Press.

Huet, G. 1980. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM 27,* 4, 797–821.

Kreowski, H.-J. and Valiente, G. 2000. Redundancy and subsumption in high-level

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

replacement systems. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, H. G. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 1765. Springer-Verlag, 215–227.

Lam, E. S. and Sulzmann, M. 2006. Towards Agent Programming in CHR. In *CHR '06: Proc. 3rd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. K. U. Leuven, Tech. Rep. CW 452, 17–31.

Lam, E. S. and Sulzmann, M. 2008. Finally, a comparison between Constraint Handling Rules and join-calculus. In *CHR '08* (Hagenberg, Austria), T. Schrijvers, F. Raiser, and T. Frühwirth, Eds. RISC Report Series 08-10, University of Linz, Austria, 51–66.

Löwe, M. 1993. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science 109,* 1&2, 181–224.

Löwe, M. and Müller, J. 1993. *Algebraische Graphersetzung: mathematische Modellierung und Konfluenz*. Tech. Rep. 93-37, Technische Universität, Berlin.

Newman, M. 1942. On theories with a combinatorial definition of equivalence. *Annals of Mathematics 43,* 2, 223–242.

Pilozzi, P. and De Schreye, D. 2008. Termination analysis of CHR revisited. In *ICLP '08*, M. García de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, 501–515.

Plump, D. 1995. On Termination of graph rewriting. In *Graph-Theoretic Concepts in Computer Science, 21st International Workshop, WG '95, Proceedings*, M. Nagl, Ed. Lecture Notes in Computer Science, vol. 1017. Aachen, Germany, 88–100.

Plump, D. 2005. Confluence of graph transformation revisited. In *Processes, Terms and Cycles*, A. Middeldorp, V. van Oostrom, F. van Raamsdonk and R. C. de Vrijer, Eds. Lecture Notes in Computer Science, vol. 3838. Springer-Verlag, 280–308.

Pretschner, A., Slotosch, O., Aiglstorfer, E. and Kriebel, S. 2004. Model-based testing for real. *J. Software Tools for Technology Transfer (STTT) 5,* 2–3, 140–157.

Raiser, F. 2007. Graph Transformation Systems in CHR. In *Logic Programming, 23rd International Conference, ICLP 2007*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer-Verlag, Porto, Portugal, 240–254.

Raiser, F. 2008. Semi-automatic generation of CHR solvers for global constraints. In *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008*, P. J. Stuckey, Ed. Lecture Notes in Computer Science, vol. 5202. Springer-Verlag, Sydney, Australia, 588–592.

Raiser, F. 2009. Research summary: Analysing graph transformation systems using extended methods from constraint handling rules. In *Twenty-fifth International Conference on Logic Programming, ICLP*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, Pasadena, CA, 540–541.

Raiser, F., Betz, H. and Frühwirth, T. 2009. Equivalence of CHR states revisited. In *Sixth International Workshop on Constraint Handling Rules (CHR)*, F. Raiser and J. Sneyers, Eds. K. U. Leuven, Tech. Rep. CW 555, 34–48.

Raiser, F. and Frühwirth, T. 2009a. Operational equivalence of graph transformation systems. In *Sixth International Workshop on Constraint Handling Rules (CHR)*, F. Raiser and J. Sneyers, Eds. K. U. Leuven, Tech. Rep. CW 555, 49–62.

Raiser, F. and Frühwirth, T. 2009b. Strong joinability analysis for graph transformation systems in CHR. *Electronic Notes in Theoretical Computer Science – Proceedings of the Fifth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2009) 253,* 4, 91–111.

Rangel, G., Lambers, L., König, B., Ehrig, H. and Baldan, P. 2008. Behavior preservation

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press

in model refactoring using DPO transformations with borrowed contexts. In *Proceedings of ICGT '08 (International Conference on Graph Transformation)*. Lecture Notes in Computer Science, vol. 5214. Springer-Verlag, 242–256.

Sneyers, J., Van Weert, P., Schrijvers, T. and De Koninck, L. 2010. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming 10*, 1, 1–47.

Sulzmann, M., Schrijvers, T., and Stuckey, P. J. 2006. Principal type inference for GHC-style multi-parameter type classes. In *APLAS '06: Proceedings of the Fourth Asian Symposium on Programming Languages and Systems* (Sydney, Australia), N. Kobayashi, Ed. Lecture Notes in Computer Science, vol. 4279. Springer-Verlag, 26–43.

Van Weert, P. 2008. JCHR. URL: `http://www.cs.kuleuven.be/~petervw/JCHR`.

Van Weert, P., Sneyers, J. and Demoen, B. 2008. Aggregates for CHR through program transformation. In *LOPSTR '07, Revised Selected Papers* (Kongens Lyngby, Denmark), A. King, Ed. Lecture Notes in Computer Science, vol. 4915, 59–73.

Van Weert, P., Sneyers, J., Schrijvers, T. and Demoen, B. 2006. Extending CHR with negation as absence. In *CHR '06* (Venice, Italy), T. Schrijvers and T. Frühwirth, Eds. K. U. Leuven, Department of Computer Science, Tech. Rep. CW 452, 125–140.

Voets, D., Pilozzi, P. and De Schreye, D. 2008. *A New Approach to Termination Analysis of CHR*. Tech. Rep. CW 506, K. U. Leuven, Department of Computer Science, Leuven, Belgium.

Wasserthal, M. 2009. *An Extensible Platform for the Analysis of Graph Transformation Systems using Constraint Handling Rules*. Diploma thesis, Ulm University, Ulm, Germany.

https://doi.org/10.1017/S1471068410000438 Published online by Cambridge University Press