Confluence in Constraint Handling Rules: A retrospective overview

Henning Christiansen and Maja H. Kirkeby

Computer Science, Roskilde University, Denmark
`henning@ruc.dk` and `majaht@ruc.dk`

1 Introduction

Constraint Handling Rules, CHR, is a nondeterministic programming language whose programs consists of rewrite rules over program states, and being able to show confluence may be an important part of a program correctness proof. Confluence of has been studied from the first introduction of CHR [8, 9]. The first essential results on proving confluence in CHR [1–3], developed during the 1990s, were formulated with respect to a logic based semantics. This choice gave elegant proofs for terminating systems based on the subsumption principle. More recent work extends the previous methods for proving confluence to include invariants and confluence modulo equivalence. Furthermore, these results were developed for a more realistic semantics that reflects the de-facto standard implementations of CHR upon Prolog, including a correct treatment of Prolog's non-logical devices (e.g., var/1, nonvar/1, is/2) and runtime errors. In the following we give an overview of confluent results for Constraint Handling Rules, from early, fundamental results to recent extensions including state invariants and confluence modulo equivalence.

2 Preliminaries

We rephrase the following standard definitions and properties. A *transition system* $D = \langle S, \mapsto \rangle$ consists of a set of *states* S; a *transition* is an element of $\mapsto : S \to S$, written $s_1 \mapsto s_2$ or, alternatively, $s_2 \mapsfrom s_1$, and $\mapsto^*$ is the reflexive transitive closure of $\mapsto$. An *object corner* is a structure of the form $s_1 \mapsfrom s \mapsto s_2$ in which the indicated relationships hold. A system is *confluent* whenever, for all s, s_1, s_2 with $s_1 \mapsfrom^* a \mapsto^* s_2$, that s_1, s_2 are joinable, i.e. there exists a state t such that $s_1 \mapsto^* t \mapsfrom^* s_2$; and it is *locally* confluent whenever any object corner is joinable. Newman's lemma: A terminating system is confluent if and only if it is locally confluent.

Proving different types of systems confluent has been facilitated by constructions of sets of critical pairs – or *critical corners* as we refer to, including the common ancestor state. Typically, these critical corners are selected object corners of the same system (but we will relax this later) and are accompanied by a notion of *subsumption* (that may vary depending on the type of system), i.e., a given critical corner subsumes a set object corners.

The set of critical corners should satisfy the following properties[1] - whenever a critical corner is joinable, any object corner that it covers is joinable, - an object corner not covered by a critical corner is known to be joinable in some way or another (and thus referred to as trivial corners). Thus, a proof of confluence for a terminating system may a matter of 1) describe a set of critical corner, 2) show each of them joinable; we refer to this proof strategy as the

[1]E.g., as a consequence of a property called the Critical pair lemma, that we no not exactly rephrase here.

subsumbtion principle. Ideally a set of critical corners is finite, but this may not always be the case.

3 Constraint Handling Rules

Constraint Handling Rules [8, 10, 11], CHR, is a nondeterministic programming language based on rewriting rules. It was originally designed for constraint solving but has become important as a general-purpose language for representing knowledge and expressing algorithms in a high-level fashion; today it is applied in many areas, for instance analysis of types, multi-agent systems, scheduling, and abductive reasoning; see, e.g., [12] for an overview. While most common CHR implementations are deterministic rather than nondeterministic, it is still useful to consider CHR programs as nondeterministic as it allows the programmer to disregard the execution model of the specific implementation.

CHR relates to the logic programming tradition; constraints are first-order atoms and the language has a declarative semantics [11] based on a logical reading of the rules. CHR programs consist of (a finite set of) guarded rewrite rules over multi-sets of constraints, called *constraint stores*.

Example 1 ([4, 5]). The following CHR program, consisting of a single rule (without guard), collects a number of separate items into a set represented as a list of items.

```
set(L), item(A) <=> set([A|L]).
```

This rule will apply repeatedly, replacing constraints matched by the left hand side by those indicated to the right, there exists a substitution from the rule constraints to the constraint store constraints. The query

```
?- item(a), item(b), set([]).
```

may lead to two different final states, {`set([a,b])`} and {`set([b,a])`}, both representing the same set.

There are three rule types in CHR; the one in the example above is a *simplification* rule that replaces constraints by new constraints; another rule type called *propagation* rule adds new constraints (such a rule is written using a different arrow '`==>`'); and the last type, called a *simpagation*, is a generalization of these two. CHR applies a bookkeeping mechanism to avoid trivial looping that otherwise arises with propagation rules. This is easily incorporated into a formal semantics, but for simplicity we have decided to ignore this. For an in-depth introduction of all rule types, see e.g. [11].

There are two sorts of constraints user-defined that are those appearing in the head of the rules, and built-in constraints. The exact set of available built-in constraints and the modeling of their meaning varies with the CHR semantics of interest. Several CHR semantics have been proposed [1–7, 11]. The semantics used for confluence considerations has traditionally only allowed logical built-ins [1–3, 6, 7, 11], but a recent semantics [4, 5] reflects the de-facto standard implementations of CHR upon Prolog, including a correct treatment of Prolog's non-logical devices (e.g., var/1, nonvar/1, is/2) and runtime errors.

The shape of a CHR state varies with the semantics: in a simple[2] logic-based semantics, a state is of the form $\langle S, B\rangle$ where S is a constraint store and B is a built-in store containing a conjunction of built-ins where each built-in has a logical meaning; and in a Prolog-based

[2]When states were introduced originally [1–3], they contained several extra state components, but these extra components were shown redundant [4, 5].

semantics, a state consists of a constraint store S and each built-in has an operational meaning producing a special substitution that is applied to S. For instance t_1>t_2 evaluates to an empty substitution if t_1 and t_2 are ground arithmetic expressions with values v_1 and v_2, and $v_1 > v_2$ holds, it evaluates to a *failure* substitution if instead $v_1 \not> v_2$, and to an *error* substitution, otherwise. Applying an *error* (a *failure*) substitution to a state change the state to a special state, namely an *error*-state (a *failure*-state); if such a state is reached no further transformations are allowed; therefore, the Prolog-based semantics is sensitive to the execution order of built-ins.

A built-in constraint may occur in the constraint store either introduced in the start query or by a rule body, and in rule-guards; user-defined constraints may not occur in the guards and built-in constraints may not occur on the rules left-hand sides. In the logic based semantics, built-ins in the constraint store are transferred to the built-in store, and in the Prolog based semantics they are evaluated by their operational meaning and the state is updated accordingly. For instance, considering the logic based semantics $\langle\{a{=}X, p(X)\}, \mathit{true}\rangle$ is updated to $\langle\{p(X)\}, X{=}a\rangle$ and considering instead the Prolog based semantics $\langle\{a{=}X, p(X)\}\rangle$ is transformed to $\langle\{p(a)\}\rangle$, see, e.g., [11] and respectively [5] for definitions.

A guard is a sequence of built-in constraints[3]; in the above example there are no guards, but the following example includes rule-guards on the right-hand side, e.g. (`X>0 |`). For the logic based semantics a rule may be applied if the built-in store implies the guard; and in the Prolog based semantics it may be applied if a rule-guard evaluates to neither failure nor an error, and it does not instantiate existing constraint store variables.

Example 2. Consider the following CHR program with four rules, r_1–r_4.

r_1:	`p(X) <=> q(X)`	r_3:	`q(X) <=> X>0	r(X)`
r_2:	`p(X) <=> r(X)`	r_4:	`r(X) <=> X≤0	q(X)`

In both semantics r_3 applies to `q`(n) constraints when n is a positive number, e.g., `q(1)`. Under the Prolog based semantics the rule cannot apply to the state $\langle\{$`q(X)`$\}\rangle$ since the variable `X` causes the guard `X>0` to result in an error. Under the logic based semantics the r_3 may transform a state $\langle\{$`q(X)`$\}$, `X>2`$\rangle$ to $\langle\{$`r(X)`$\}$, `X>2`$\rangle$ because `X>0` is a logical consequence of `X>2`, whereas r_3, for instance, does not transform the state $\langle\{$`q(X)`$\}, \mathit{true}\rangle$.

4 Confluence in CHR

The results on confluence for CHR are similar to those for term rewriting systems; critical corners appear when two instances of rules can apply to overlapping constraints in the constraint store, see, e.g., following example.

Example 3 (Ex. 1 continued). The `set`-program of Example 1 is not confluent under neither semantics[4], as both critical corners

$$\begin{tikzcd}
\texttt{set([X1|L]),item(X2)\}} & \texttt{\{set([X|L1]),set(L2)\}} \\
\texttt{\{item(X1),set(L),item(X2)\}} \arrow[u, mapsto] \arrow[d, mapsto] & \langle\texttt{\{set(L1),item(X),set(L2)\}} \arrow[u, mapsto] \arrow[d, mapsto] \\
\texttt{\{item(X1),set([X2|L])\}} & \texttt{\{set(L1),set([X|L2])\}}
\end{tikzcd}$$

are not joinable. Note that the built-in store would be *true* under the logic based semantics.

[3]In the logic based semantics it is seen as a conjunction. In the Prolog based semantics they are evaluated from left to right, each influencing the whole state including the subsequent series of the built-ins; if the guard evaluates to failure or error these updates are discarded, see [5] for details.

[4]The built-in store under the logic based semantics is *true* for all indicated states.

Example 4 (Ex. 2 continued). The program of Example 2 is not confluent under neither semantics[4], as its single critical corner $\{\texttt{q(X)}\} \hookleftarrow \{\texttt{p(X)}\} \mapsto \{\texttt{r(X)}\}$ is not joinable.

When a critical corner is formed by rules with guards, their satisfaction is incorpotated into the common ancestor state, which works nicely under the logic based semantics when only logical built-ins are assumed. Here subsumption of an object corner by a critical corner is defined by applying substitution and adding more constraints; the inherent monotonicity ensures joinabilty of any object corner subsumed by a critical corner.

However, the subsumption principle as explained so far as well as the use of the logic based semantics cannot handle not-logical built-in predicates that are available – and extensively used in practice – in standard implementations of CHR. Consider, for example, the CHR rule `p(X) <=> var(X) | q(X)`, whose guard is consists of Prolog's test for whtter its argument is an uninstantiated variable. While the rule may apply to a state containing `p(X)`, it does not apply to a more specific state containing `p(1)`. The other way round, if the guard is instead `nonvar(X)`, the rule may apply to a lot of subsumed instanced with `p(1)`, `p(2)`, ..., but this cannot be "verified" by investigating a most general state including `p(X)` to which the `nonvar` version of the rule does not apply.

In order to restore a subsumption principle aiming at finite proofs, we have taken the consequence in our own work to describe critical corners in a different system with higher expressibility than the object system. In the informal example considered above we can formally characterize — as a meta-level state – expressions like"$\texttt{p}(x)$ where x is a variable", as well as "... x is a constant", and perform meta-level transitions. As it is shown below, the use of such a meta-level representation is also a powerful tool when confluence under invariant and/or modulo equivalence, which otherwise has been problematic, even for the logical subset of CHR under a logic-based semantics.

5 Invariants and modulo equivalence

It can be argued that invariants and confluence modulo equivalence are important from a practical point of view. In this section we give definitions and examples and later we consider how to prove the properties. Most programs are developed with a particular set of initial queries in mind, which reduces the set of reachable states. In 2007, Duck et al. [7] suggested to take such an induced invariant into account and, thus, make a much larger class of programs confluent.

Definition 1. A set I is a state *invariant* for a relation $\mapsto$ if $x \in I \wedge x \mapsto y$ implies $y \in I$.

Such an invariant may be *induced* by a set of reachable states from a set of (initial) states Q, i.e. $I = \{s' \mid s \in Q \wedge s \mapsto^* s'\}$.

Example 5 (Ex. 1 continued). The *set*-program reflects a tacitly assumed state invariant: only one `set`-constraint is allowed. If we open up for a query such as

```
?- item(a), item(b), set([]), set([c]).
```

we obtain a collection of different answers, representing different ways of partitioning {`a`, `b`, `c`} into two sets. However, this may not be intended, and the relevant invariant $I_{\texttt{set}}$ must specify that a state must include at most one `set`/1 constraint and a series of `item`/1 constraints.

Definition 2. A relation $\mapsto$ is *observable confluent* (under invariant I) if and only if $\forall x, y, y' \in I \colon y' \hookleftarrow^* x \mapsto^* y' \Rightarrow \exists z \in I \colon y' \mapsto^* z \hookleftarrow^* y'$. We may write *I-observable confluent* meaning observable confluent (under invariant I).

Example 6 (Ex. 2 continued). Consider the program of Example 2 together with an invariant $I_{>0}$ induced by initial states with a single atom `p(X)` where n is a positive number (not a variable). The program is $I_{>0}$-observable confluent, since each forked state $\mathtt{q}(n) \leftarrowtail \mathtt{p}(n) \mapsto \mathtt{r}(n)$ is joinable by rule r_3: $\mathtt{r}(n) \mapsto \mathtt{q}(n)$.

Confluence modulo equivalence is a generalization where forked states must reach equivalent states, rather than a common state. For instance, a program may produce redundant data structures such as representing sets as lists, and the equivalence states that the order of the elements does not matter. Confluence modulo equivalence was first considered for CHR in 2014 by Christiansen and Kirkeby [4].

Definition 3. A relation $\mapsto$ is *confluent modulo an equivalence* $\approx$ if and only if $\forall x, y, x', y' : y' \leftarrowtail^* x' \approx x \mapsto^* y' \Rightarrow \exists z, z' : y' \mapsto^* z' \approx z \leftarrowtail^* y'$.

Example 7 (Ex. 1, 5 continued). The *set*-program is supposed to produce one set representation, and we introduce a state equivalence $\approx_{\mathtt{set}}$ reflecting the redundant data structures. Two states are equivalent if they have the same `item`-constraints and their respective `set`-constraint $\mathtt{set}(L_1)$ and $\mathtt{set}(L_2)$ are such that L_1 and L_2 are permutations of each other.

We generalize confluence modulo equivalence and observable confluence as follows.

Definition 4. A relation $\mapsto$ is *I-observable confluent modulo an equivalence* $\approx$ if and only if $\forall x, y, x', y' \in I : y' \leftarrowtail^* x' \approx x \mapsto^* y' \Rightarrow \exists z, z' \in I : y' \mapsto^* z' \approx z \leftarrowtail^* y'$.

Both observable confluence, confluence modulo equivalence and classic confluence are special cases of this definition.

Huet [14] provided a pair of local properties for showing terminating programs confluent modulo equivalence; we extend these with an invariant as follows.

Definition 5. A rewriting system $\mapsto$ is *locally I-observable confluent modulo* $\approx$ if and only if it has the following α- and β-properties.

$$\begin{array}{llcll}
\alpha: & \forall x, y, y' \in I: \quad y' \leftarrowtail x \mapsto y' & \Rightarrow & \exists z, z' \in I: & y' \mapsto^* z' \approx z \leftarrowtail^* y' \\
\beta: & \forall x, y, y' \in I: \quad y' \approx x \mapsto y' & \Rightarrow & \exists z, z' \in I: & y' \mapsto^* z' \approx z \leftarrowtail^* y'
\end{array}$$

We refer to structures of the form $y' \leftarrowtail x \mapsto y'$ as α-corners and those of the form $y' \approx x \mapsto y'$ as β-corners.

Theorem 1 (obs. confl. mod. equivalence). *A terminating relation $\mapsto$ is I-observable confluent modulo $\approx$ if and only if it is locally I-observable confluent modulo $\approx$.*

In the special cases where equivalence is '=', the β-property trivially holds and when, furthermore, the invariant is unrestrictive the α-property reduces to local confluence.

Both the invariant and the equivalence relation may be tailored for the individual program. By nature, they are meta level properties that in general cannot be expressed in its own system: the state itself is implicit and properties such as groundness (or certain arguments bound to be variables) cannot be expressed in a logic-based semantics for CHR.

6 Proving observable confluence

As mentioned, observable confluence was introduced by Duck et al. [7]. They suggested methods of proving this property for logic CHR programs using a logic based semantics and as a direct continuation of the logic subsumption principle.

Firstly, they construct the set of critical corners based from the program rules as explained in Section 4. Typically, the states in these corners do not satisfy the invariant (a rule typically includes variables, contradicting groundness), and the next step is to characterize a set of "minimal extensions" of each critical corner such that 1) the states of such an extension satisfies the invariant, 2) the set of all such extensions subsumes (by substitution and adding constraints) all non-trivial object corners. Proving observable confluence amounts to show joinability af such extension, using the standard transition relation for CHR.

There are two problems in this approach, first of all there is no formal representation of the invariant that allows to take it into account when reasoning formally about joinability, and – more importantly – as also noticed by Duck et al. [7], quite often there is an infinite number of such extensions. This happens even for an intuitively simple invariant such as requiring ground states. We can demonstrate this phenomenona for the program of Example 2.

Example 8. Consider the program of Example 2; it is not confluent as its single critical corner $\texttt{q(X)} \leftarrowtail \texttt{p(X)} \mapsto \texttt{r(X)}$ is not joinable (the built-in store is always *true* and thus omitted). However, adding an invariant "reachable from an initial state $\texttt{p(}n\texttt{)}$ where n is an integer" makes it confluent. We indicate the smallest set of corners found by minimal extensions of the critical corner; the dotted transitions prove each of them joinable:

$$\begin{tikzcd}[column sep=small]
\cdots & & \texttt{p(-1)} \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & & & \texttt{p(0)} \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & & & \texttt{p(1)} \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & & & \texttt{p(2)} \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & & \cdots \\
& \texttt{q(-1)} & & \texttt{r(-1)} \arrow[ll, dotted, "r_4"'] & \texttt{q(0)} & & \texttt{r(0)} \arrow[ll, dotted, "r_4"'] & \texttt{q(1)} \arrow[rr, dotted, "r_3"] & & \texttt{r(1)} & \texttt{q(2)} \arrow[rr, dotted, "r_3"] & & \texttt{r(2)} &
\end{tikzcd}$$

This set is exactly the set of all non-trivial object level corners. These corners and their proofs of joinability obviously fall in two groups of similar shapes, but there is no way to construct a finite set (of, say, one or two elements) of critical corners in CHR, that covers all object corner.

To avoid this problem, Christiansen and Kirkeby [15] suggests to describe critical corners in a more powerful meta-language rather than using CHR itself, inspired by earlier work on meta-programming in logic programming. Each term of CHR is here named by a ground term, specifically, variables named by ground constants. A variable in such a term is thus a meta-variable, which may be covered by a meta-level constraint. For example, the meta-level term $\texttt{p(}x\texttt{)}$ WHERE $\texttt{type(const,}X\texttt{)}$ subsumes all object level (=CHR) atoms whose predicate is $\texttt{p}/1$ and whose argument is a constant, i.e., `p(a)`, `p(b)`, ..., `p(1)`, ... The authors define a notion of abstract similation making precise what it means for meta-level transitions and corners to subsume[5] sets of object level transitions and corners. Built-in predicates are reflected at the meta-level, such that, say $n \widehat{<} 0$ restricts n to names of terms t that satisfy the object level condition $t < 0$, i.e., n is limited names of number constrants less that zero.

Example 9. (Continuing Ex. 2, 8) The invariant is formalized at the meta-level as states of the form $\langle\{\mathit{pred}(n)\}, \mathit{true}\rangle$ WHERE $\texttt{type(int,}n\texttt{)}$ where *pred* is one of `p`, `q` and `r`. Below is shown the two joinable critical meta-level corners that can be shown to subsume all non-trivial object level corners.

[5] [15] use the terminology of a meta-level term *covering* object level notions.

$$\begin{tikzcd}
 & \{\texttt{p(X)}\}\ \textsc{where}\ \texttt{type(int,X), X}\hat{\geq}\texttt{0} \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & \\
\{\texttt{q(X)}\}\ \textsc{where}\ \texttt{type(int,X), X}\hat{\geq}\texttt{0} & & \{\texttt{r(X)}\}\ \textsc{where}\ \texttt{type(int,X), X}\hat{\geq}\texttt{0} \arrow[ll, dotted, "r_4"']
\end{tikzcd}$$

$$\begin{tikzcd}
 & \{\texttt{p(X)}\}\ \textsc{where}\ \texttt{type(int,X), X}\hat{<}\texttt{0)} \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & \\
\{\texttt{q(X)}\}\ \textsc{where}\ \texttt{type(int,X), X}\hat{<}\texttt{0} \arrow[rr, dotted, "r_3"] & & \{\texttt{r(X)}\}\ \textsc{where}\ \texttt{type(int,X), X}\hat{<}\texttt{0}
\end{tikzcd}$$

This example illustrates an additional technique called splitting: First a meta-level corner is produced in the classical way, considering how rules can overlap; this yields a common meta-level ancestor state $\texttt{p}(x)$ WHERE $\texttt{type(int,}x\texttt{)}$ and the other states as above containing $\texttt{q}(x)$, resp. $\texttt{r}(x)$. This meta-level corner is in itself not joinable as no single rule can apply, but turning it into two meta-level corners, each joinable and together subsuming the same set of object-level corners, proves observable confluence.

7 Proving Confluence modulo equivalence

Confluence modulo equivalence has been studied since the first half of the 20th century in a variety of contexts; see, e.g., [5, 14] for an overview. It was introduced and motivated for CHR by [4], also arguing that invariants are important for specifying meaningful equivalences. An in-depth theoretical analysis, including the use of the ground representation, is given by [5] in relation the Prolog-related semantics mentioned above.

To show confluence modulo equivalence for terminating CHR programs, two types of critical corners must be constructed: *critical α-corners* which are the standard critical corners constructed by rule overlap as above, and *critical β-corners* of the form $y' \approx x \mapsto y'$, cf. Definition 5. As before the critical β-corners must subsume all non-trivial object-level β-corners. The meta-level language described above is also suitable for describing state equivalences [15].

Example 10 (Ex. 1 continued). The `set`-program of Example 1 is observable confluent modulo $\approx_{\texttt{set}}$ (Ex. 7) under invariant $I_{\texttt{set}}$ (Ex. 5) since the critical α-corner with two `set`-constraints does not subsume $I_{\texttt{set}}$ corners and both the other critical α-corner and the critical β-corner are joinable modulo $\approx_{\texttt{set}}$, see below. The meta-level constraint `its`/1 constrains its argument to a set of `item`-constraints and `perm`/2 constrains the arguments to a pair of permuted lists.
α-corner:

$$\begin{tikzcd}
\{\texttt{set([X1|L]),item(X2)}\} \uplus C\ \textsc{where}\ \texttt{its}(C) \arrow[r, mapsto] & \{\texttt{set([X2,X1|L])}\} \uplus C\ \textsc{where}\ \texttt{its}(C) \\
\{\texttt{item(X1),set(L),item(X2)}\} \uplus C\ \textsc{where}\ \texttt{its}(C) \arrow[u, mapsto] \arrow[d, mapsto] & \wr\wr \\
\{\texttt{item(X1),set([X2|L])}\} \uplus C\ \textsc{where}\ \texttt{its}(C) \arrow[r, mapsto] & \{\texttt{set([X1,X2|L])}\} \uplus C\ \textsc{where}\ \texttt{its}(C)
\end{tikzcd}$$

β-corner:

$$\begin{tikzcd}
\{\texttt{set([L2]),item(X)}\} \uplus C\ \textsc{where}\ \texttt{perm(L1,L2)} \wedge \texttt{its}(C) \arrow[r, mapsto] & \{\texttt{set([X|L2])}\} \uplus C\ \textsc{where}\ \texttt{perm(L1,L2)} \wedge \texttt{its}(C) \\
\{\texttt{set([L1]),item(X)}\} \uplus C\ \textsc{where}\ \texttt{perm(L1,L2)} \wedge \texttt{its}(C) \arrow[u, phantom, "\wr\wr"] \arrow[d, mapsto] & \rightsquigarrow \\
\{\texttt{set([X|L1])}\} \uplus C\ \textsc{where}\ \texttt{perm(L1,L2)} \uplus C \wedge \texttt{its}(C) &
\end{tikzcd}$$

A recent paper [13] attempts to handle (observable) confluence modulo equivalence within the logic-based semantics, along the lines of Duck et al [7]. However, this implies the mentioned problems of infinitely many proof cases, which seems to be inherent in relying on pure logic-based subsumption without having the enhanced expressibility provided by a suitable meta-level representation.

8 Future work

We have given an overview of classic and recent results for confluence in CHR. The classic results provide a theoretical foundation and the recent results on observable confluence and modulo equivalence points towards more practical applications of these notions in a programming context.

There exist methods for automatic check of confluence for CHR [16] in a strictly logical setting, and in our own work we are developing similar methods for automatic or semi-automatic proofs of observable confluence modulo equivalence. Naturally, invariants and state equivalences may involve undecidability.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *CP*, pages 252–266, 1997.

[2] S. Abdennadher, T. W. Frühwirth, and H. Meuss. On confluence of Constraint Handling Rules. In *CP*, volume 1118 of *LNCS*, pages 1–15. Springer, 1996.

[3] S. Abdennadher, T. W. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.

[4] H. Christiansen and M. H. Kirkeby. Confluence modulo equivalence in constraint handling rules. In *LOPSTR 2014*, volume 8981 of *LNCS*, pages 41–58, 2015.

[5] H. Christiansen and M. H. Kirkeby. On proving confluence modulo equivalence for constraint handling rules. *Formal Aspects of Computing*, 29(1):57–95, 2017.

[6] G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *ICLP 2004*, volume 3132 of *LNCS*, pages 90–104. Springer, 2004.

[7] G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable confluence for Constraint Handling Rules. In *ICLP*, volume 4670 of *LNCS*, pages 224–239. Springer, 2007.

[8] T. W. Frühwirth. User-defined constraint handling. In *ICLP*, pages 837–838. MIT Press, 1993.

[9] T. W. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, May 16 - 20, 1994, Selected Papers*, volume 910 of *LNCS*, pages 90–107. Springer, 1994.

[10] T. W. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[11] T. W. Frühwirth. *Constraint Handling Rules.* Cambridge Uni. Press, 2009.

[12] T. W. Frühwirth. Constraint handling rules - what else? In N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, and D. Roman, editors, *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings*, volume 9202 of *Lecture Notes in Computer Science*, pages 13–34. Springer, 2015.

[13] D. Gall and T. Frühwirth. Confluence modulo equivalence with invariants in Constraint Handling Rules. In *FLOPS 2018*, 2018. To appear.

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

[15] M. H. Kirkeby and H. Christiansen. Confluence of CHR revisited: invariants and modulo equivalence. *CoRR*, 2018. *(submitted to an international symposium).*

[16] J. Langbein, F. Raiser, and T. W. Frühwirth. A state equivalence and confluence checker for CHRs. In *Proc. Int'l Workshop on Constraint Handling Rules*, Report CW 588, pages 1–8. Katholieke Universiteit Leuven, Belgium, 2010.