Confluence of CHR Revisited

Invariants and Modulo Equivalence

Christiansen, Henning; Kirkeby, Maja Hanne

Published in:
Logic-Based Program Synthesis and Transformation

DOI:
10.1007/978-3-030-13838-7_6

Publication date:
2019

Document Version
Peer reviewed version

Citation for published version (APA):
Christiansen, H., & Kirkeby, M. H. (2019). Confluence of CHR Revisited: Invariants and Modulo Equivalence. In F. Mesnard, & P. J. Stuckey (Eds.), *Logic-Based Program Synthesis and Transformation: 28th International.Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers* (Vol. 11408, pp. 94-111). Springer. https://doi.org/10.1007/978-3-030-13838-7_6

Confluence of CHR revisited: invariants and modulo equivalence⋆ ⋆⋆

Henning Christiansen and Maja H. Kirkeby

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

Abstract. Abstract simulation of one transition system by another is introduced as a means to simulate a potentially infinite class of similar transition sequences within a single transition sequence. This is useful for proving confluence under invariants of a given system, as it may reduce the number of proof cases to consider from infinity to a finite number. The classical confluence results for Constraint Handling Rules (CHR) can be explained in this way, using CHR as a simulation of itself. Using an abstract simulation based on a ground representation, we extend these results to include confluence under invariant and modulo equivalence, which have not been done in a satisfactory way before.

Keywords: Constraint Handling Rules, Confluence, Confluence modulo equivalence, Invariants, Observable confluence

1 Introduction

Confluence of a transition system means that any two alternative transition sequences from a given state can be extended to reach a common state. Proving confluence of nondeterministic systems may be important for correctness proofs and it anticipates parallel implementations and application order optimizations. Confluence modulo equivalence generalizes this so that these "common states" need not be identical, but only equivalent according to an equivalence relation. This allows for redundant data representations (e.g., sets as lists) and procedures that search for an optimal solution to a problem, when any of two equally good solutions can be accepted (e.g., the Viterbi algorithm analyzed for confluence modulo equivalence in [8]).

We introduce a notion of abstract simulation of one system, the object system, by another, the meta level system, and show how proofs of confluence (under invariant, modulo equivalence) for an object system may be expressed within a meta level system. This may reduce the number of proof cases to be considered,

⋆ This work is supported by The Danish Council for Independent Research, Natural Sciences, grant no. DFF 4181-00442.

⋆⋆ This paper has been published in Proc. LOPSTR 2018; Springer LNCS 11408, 2019; pp. 94-111.

often from infinity to a finite number. We apply this to the programming language of Constraint Handling Rules, CHR [14,15,16], giving a clearer exposition of existing results and extending them for invariants and modulo equivalence.

By nature, invariants and state equivalences 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 restricted to be uninstantiated variables) cannot be expressed in a logic-based semantics for CHR. Using abstract simulation we can add the necessary enhanced expressibility to the meta level, and the ground representation of logic programs, that was studied in-depth in the late 1980s and -90s in the context of meta-programming in logic (e.g., [5,19,18]), comes in readily as a well-suited and natural choice for this. The following minimalist example motivates both invariant and state equivalence for CHR.

Example 1 ([7,8]). The following CHR program, consisting of a single rule, 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 the one indicated to the right. 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. Thus, the program is not confluent, but it may be confluent modulo an equivalence that disregards the order of the list-elements. Confluence modulo equivalence still requires an invariant that excludes more than one `set`/1 constraint, as otherwise, an element may go to an arbitrary of those.

1.1 Related work

Some applications of our abstract simulations may be seen as special cases of abstract interpretation [10]. This goes for the re-formulation of the classical confluence results for CHR, but when invariants are introduced, this is not obvious; a detailed argument is given in Section 5. It is related to symbolic execution and constraint logic programming [22], where reasoning takes place on compact abstract representations parameterized in suitable ways, rather than checking multitudes of concrete instances. Bisimulation [26], which has been applied in many contexts, indicates a tighter relationship between states and transitions of two systems than the abstract simulation: when a state s_0 is simulated by an abstract state s'_0 and there is a transition $s_0 \rightarrow s_1$, bisimulation would require the existence of an abstract transition $s'_0 \rightarrow' s'_1$, which may not be case as demonstrated by Example 6.

Previous results on confluence of CHR programs, e.g., [1,2,3], mainly refer to a logic-based semantics, which is well-suited for showing program properties, but it does not comply with typical implementations [20,28] and applies only for a small subset of CHR programs. Other works [7,8] suggest an alternative operational semantics that lifts these limitations, including the ability to handle Prolog-style built-in predicates such as `var/1`, etc. To compare with earlier work and for simplicity, the present paper refers to the logic-based semantics.

As long as invariants and modulo equivalence are not considered, the logic-based semantics allows for elegant confluence proofs based on Newman's Lemma (Lemma 1, below). A finite set of critical pairs can be defined, whose joinability ensures confluence for terminating programs. Duck et al. [13] proposed a generalization of this approach to confluence under invariant, called observable confluence; no practically relevant methods were suggested, and (as the authors point out) even a simple invariant such as groundness explodes into infinitely many cases.

Confluence modulo equivalence was introduced and motivated for CHR by [7], also arguing that invariants are important for specifying meaningful equivalences. An in-depth theoretical analysis, including the use of a ground representation, is given by [8] in relation the alternative semantics mentioned above. However, it has not been related to abstract simulations, and the proposal for a detailed language of meta level constraints in the present paper is new. Repeating the motivations of [7,8] in the context of the logic-based semantics, [17] suggested to handle confluence modulo equivalence along the lines of [13], thus inheriting the problems of infinitely many proof case pointed out above.

An approach to show confluence of a transition system, by producing a mapping into another confluent system, is described by [11] and extended to confluence modulo equivalence by [23]; the relationship between such two systems is different from the abstract simulations introduced in the present paper. Confluence, including modulo equivalence, has been studied since the first half of the 20th century in a variety of contexts; see, e.g., [8,21] for overview.

1.2 Contributions

We introduce abstract simulation as a setting for proofs of confluence for general transitions systems and demonstrate this specifically for CHR. We recast classical results (without invariant and equivalence), showing that they are essentially based on a simulation of CHR's logic-based semantics by itself, and we can pinpoint, why it does not generalize for invariants (see Example 4, p. 9).

These results are extended for invariants and modulo equivalence, using an abstract simulation; it is based on a ground meta level representation and suitable meta level constraints to reason about it.

1.3 Overview

Sections 2 and 3 introduce basic concepts of confluence plus our notion of abstract simulation. Section 4 gives syntax and semantics of CHR along with a discussion of how much nondeterminism to include in a semantics used when considering confluence. Section 5 re-explains the classical results in terms of abstract simulation. Section 6 extends these results for invariants and modulo equivalence; proofs can be found in an extended report [6]. The concluding Section 7 gives a summary and explains briefly how standard mechanisms, used to prevent loops by CHR's propagation rules, can be added.

2 Basic concepts, confluence, invariants and equivalences

A *transition system* $D = \langle S, \mapsto \rangle$ consists of a set of *states* S, and a *transition* is an element of $\mapsto: S \times S$, written $s_0 \mapsto s_1$ or, alternatively, $s_1 \mapsfrom s_0$. A *transition sequence* or *path* is a chain of transitions $s_0 \mapsto s_1 \mapsto \cdots \mapsto s_n$ where $n \geq 0$; if such a path exists, we write $s_0 \overset{*}{\mapsto} s_n$. A state s_0 is *final* (or *normal form*) whenever $\nexists s_1\ s_0 \mapsto s_1$, and D is *terminating* whenever every path is finite. To anticipate the application for logic programming systems, a given transition system may have a special final state called *failure*.

An *invariant* I for $D = \langle S, \mapsto \rangle$ is a subset $I \subseteq S$ such that

$$s_0 \in I \wedge s_0 \mapsto s_1 \quad \Rightarrow \quad s_1 \in I.$$

We write a fact $s \in I$ as $I(s)$ and refer to s as an I *state*. The *restriction of* D *to* I is the transition system $\langle I, \overset{I}{\mapsto} \rangle$ where $\overset{I}{\mapsto}$ is the restriction of $\mapsto$ to I. A set of *allowed initial states* $S' \subseteq S$ defines an invariant of those states *reachable* from some $s \in S'$, i.e., $\text{reachable}(S') = \{s' \mid s \in S' \wedge s \overset{*}{\mapsto} s'\}$. A *(state) equivalence* is an equivalence relation over S, typically denoted $\approx$. In the context of an invariant I, the relations $\approx$ and $\mapsto$ are understood to be restricted to I.

The following α and β corners[1] were introduced in [7,8], being implicit in [21]. An α *corner* is a structure $s_1 \mapsfrom s_0 \mapsto s_2$, where $s_0, s_1, s_2 \in S$ and the indicated relationships hold; s_0 is called a *common ancestor* and s_1, s_2 *wing* states. A β *corner* is a structure $s_1 \approx s_0 \mapsto s_2$, where $s_0, s_1, s_2 \in S$ and the indicated relationships hold. In the context of an invariant I, the different types of corners are defined only for I states.

Two states s_1, s_2 are *joinable (modulo* $\approx$*)* whenever there exist paths $s_1 \overset{*}{\mapsto} s_1'$ and $s_2 \overset{*}{\mapsto} s_2'$ with $s_1' = s_2'$ ($s_1' \approx s_2'$). A corner $s_1\ \mathit{Rel}\ s_0 \mapsto s_2$ is *joinable (modulo* $\approx$*)* when s_1, s_2 are joinable (modulo $\approx$); $\mathit{Rel} \in \{\mapsfrom, \approx\}$.

A transition system $D = \langle S, \mapsto \rangle$ is *confluent (modulo* $\approx$*)* whenever

$$s_1 \overset{*}{\mapsfrom} s_0 \overset{*}{\mapsto} s_2 \quad \Rightarrow \quad s_1 \text{ and } s_2 \text{ are joinable (modulo } \approx).$$

It is *locally confluent* (modulo equivalence $\approx$) whenever all its α (α and β) corners are joinable. The following properties are fundamental.

Lemma 1 (Newman [25]). *A terminating transition system (under invariant* I*) is confluent if and only if it is locally confluent.*

Lemma 2 (Huet [21]). *A terminating transition system (under invariant* I*) is confluent modulo* $\approx$ *if and only if it is locally confluent modulo* $\approx$*.*

[1] In recent literature within term rewriting, the terms peaks and cliffs have been used for α and β corners, respectively.

These properties reduce proofs of confluence (mod. equiv.) for terminating systems to proofs of the simpler property of local confluence (mod. equiv.), but still, this may leave an infinite number of corners to be examined.

3 Abstract Simulation

Consider two transition systems, $D_O = \langle S_O, \mapsto_O \rangle$ and $D^M = \langle S^M, \mapsto^M \rangle$, referred to as *object* and *meta level* systems. A *replacement* is a (perhaps partial) function $\rho\colon S^M \to S_O$; the application of ρ to some $s \in S^M$ is written $s\rho$. For any structure $f(s_1, \dots s_n)$ with states $s_1, \dots s_n$ of D^M (a transition, a tuple, etc.), replacements apply in a compositional way, $f(s_1, \dots s_n)\rho = f(s_1\rho, \dots s_n\rho)$. For a family of replacements $P = \{\rho_i\}_{i \in \mathit{Inx}}$, the *covering* (or *concretization*) of a structure $f(s_1, \dots s_n)$ is defined as

$$[\![f(s_1, \dots s_n)]\!]^M_O = \{f(s_1, \dots s_n)\rho \mid \rho \in P\}.$$

Notice that P is left implicit in this notation, as in the context of given object and meta level systems, there will be one and only one replacement family.

Definition 1. *An* abstract simulation of D_O by D^M *with possible invariants* I_O*, resp.,* I^M*, and equivalences* $\approx_O$*, resp.,* $\approx^M$*, is defined by a family of replacements* $P = \{\rho_i\}_{i \in \mathit{Inx}}$ *which satisfies the following conditions.*

$$\begin{array}{rcl} s_0 \mapsto^M s_1 & \Rightarrow & \forall \rho \in P\colon\ s_0\rho \mapsto_O s_1\rho \ \vee\ s_0\rho = s_1\rho \\ I^M(s) & \Rightarrow & \forall \rho \in P\colon\ I_O(s\rho) \\ s_0 \approx^M s_1 & \Rightarrow & \forall \rho \in P\colon\ s_0\rho \approx_O s_1\rho \end{array}$$

Notice that an abstract simulation does not necessarily cover all object level states, transitions, etc.

Example 2. Let $A = \{a_1, a_2, \dots\}$, $B = \{b_1, b_2, \dots\}$ and $C = \{c_1, c_2, \dots\}$ be sets of states, and O and M the following transition systems.

$$\begin{array}{rcl} O & = & \langle A \cup B \cup C, \{a_i \mapsto_O b_i \mid i = 1, 2, \dots\} \cup \{a_i \mapsto_O c_i \mid i = 1, 2, \dots\}\rangle \\ M & = & \langle \{a, b, c\}, \{a \mapsto_M b, a \mapsto_M c\}\rangle \end{array}$$

Assume equivalences $b \approx^M c$ and $b_i \approx_O c_i$, for all i. Then the family of replacements $P = \{\rho_i\}_{i=1,2,\dots}$, where $a\rho_i = a_i$, $b\rho_i = b_i$ and $c\rho_i = c_i$, defines a simulation of O by M. It appears that O and M are not confluent, cf. the non-joinable corners $b_1 \leftarrow\!\shortmid_O a_1 \mapsto_O c_1$ and $b \leftarrow\!\shortmid_M a \mapsto_M c$, but both are confluent modulo $\approx_O$ ($\approx^M$).

A meta level structure m *covers* an object structure k whenever $k \in [\![m]\!]^M_O$. When $[\![m]\!]^M_O = \emptyset$, m is *inconsistent*. When $[\![m']\!]^M_O \subseteq [\![m]\!]^M_O$, m' is a *substate*/*subcorner*, etc. of m, depending on the inherent type of m. When D_O and D^M both include *failure*, it is required that $[\![\mathit{failure}]\!]^M_O = \{\mathit{failure}\}$. A given meta level state S is *mixed* whenever $[\![S]\!]^M_O$ includes both *failure* and non-*failure* states. Transitions are only allowed from consistent and neither failed nor mixed states.

The following is a consequence of the definitions.

Lemma 3. *An object level corner, which is covered by a joinable (mod. equiv.) meta level corner, is joinable (mod. equiv.).*

When doing confluence proofs, we may search for a small set of *critical* meta level corners,[2] whose joinability guarantees joinability of any object level corner, i.e., any other object level corner not covered by one of these is seen to be joinable in other ways. For term rewriting systems, e.g., [4], and previous work on CHR, such critical sets have been defined by explicit constructions.

We introduce a mechanism for splitting a meta level corner Λ into a set of corners, which together covers the same set of object corners as Λ. This is useful when Λ in itself is not joinable, but each of the new corners are. In some cases, splitting is necessary for proving confluence under an invariant as shown in Section 5 and exemplified in Examples 4 and 6.

Definition 2. *Let s be a meta level state (or corner). A set of states (or corners) $\{s_i\}_{i\in Inx}$ is a* splitting *of s whenever $\bigcup_{i\in Inx}[\![s_i]\!]^M_O = [\![s]\!]^M_O$. A corner (set of corners) is* split joinable *(mod. equiv.) if it (each of its corners) is joinable (mod. equiv.), inconsistent, or has a splitting into a set of split joinable (mod. equiv.) corners.*

Corollary 1. *An object level corner, which is covered by a split joinable (mod. equiv.) meta level corner, is joinable (mod. equiv.).*

4 Constraint Handling Rules

Most actual implementations of CHR are fully deterministic, i.e., for a given query, there is at most one answer state (alternatively, the program is nonterminating). In this light, it may be discussed whether confluence is an interesting property, and if so, to what extent the applied semantics should be nondeterministic. Our thesis is the following: choice of next constraints to be tried and which rule to be used should be nondeterministic. Thus a confluent program can be understood by the programmer without considering the detailed control mechanisms in the used implementation; this also anticipates parallel implementations. We see only little interest in considering confluence for the so-called refined CHR semantics [12] in which only very little nondeterminism is retained.

Similarly to [7,8], we remove w.l.o.g. two redundancies from the logic-based semantics [1,16]: global variables and the two-component constraint store.

- Global variables are those in the original query. Traditionally they are kept as a separate state-component, such that values bound to them can be reported to the user at the end. The same effect can be obtained by a constraint `global`/2 that does not appear in any rule, but may be used in the original query: writing `?- p(X)` as `?- p(X), global('X',X)`, means that the value

[2] In the literature, the term *critical pair* is used for the pair of wing states of our critical corners.

of the variable named `'X'` can be read out as the second argument of this constraint in a final state.
– We avoid separating the constraint store into query and active parts, as the transition sequences with or without this separation are essentially the same.

4.1 Syntax

Standard first-order notions of variables, terms, predicates atoms, etc. are assumed. Two disjoint sets of *constraint predicates* are assumed, *user constraints* and *built-in constraints*; the actual set of built-ins may vary depending on the application. We use the generalized simpagation form [16] to capture all rules of CHR. A *rule* is a structure of the form

$$H_1 \backslash H_2 \texttt{ <=> } G \,|\, C$$

where $H_1 \backslash H_2$ is the *head* of the rule, H_1 and H_2 being sequences, not both empty, of user constraints; G is the *guard* which is a conjunction of built-in constraints; and C is the *body* which is a sequence of constraints of either sort. When H_2 is empty, the rule is a *simplification*, which may be written H_1 `<=>` $G \,|\, C$; when H_2 is empty, it is a *propagation*, which may be written H_2 `==>` $G \,|\, C$; any other rule is a *simpagation*; when $G = \mathit{true}$, $(G\,|)$ may be left out. The *head variables* of a rule are those appearing in the head, any other variable is *local*. The following notion is convenient when defining the CHR semantics and its meta level simulation.

Definition 3. *A* pre-application *of a rule* $r = (H_1 \backslash H_2 \texttt{<=>} G \,|\, C)$ *is of the form* $(H_1' \backslash H_2' \texttt{<=>} G' \,|\, C')\sigma$ *where* $r' = (H_1' \backslash H_2' \texttt{<=>} G' \,|\, C')$ *is a variant of* r *with fresh variables and* σ *is a substitution to the head variables of* r'*, where, for no variable* x*,* $x\sigma$ *contains a local variable of* r'*.*

The operator $\uplus$ refers to union of multisets, so that, e.g., $\{a,a\} \uplus \{a\} = \{a,a,a\}$; for difference of multisets, we use standard notation for set difference, assuming it takes into account the number of copies, e.g., $\{a,a\} \setminus \{a\} = \{a\}$.

4.2 The logic-based operational semantics for CHR

The semantics presented here is essentially identical to the one used by [1] and the so-called abstract operational semantics ω_t of [16], taking into account the simplifications explained above. Following [27], we define a state as an equivalence class, abstracting away the specific variables used and the different ways the same logical meaning can be expressed by different conjunctions of built-ins.[3] A logical theory $\mathcal{B}$ is assumed for the built-in predicates.

A *state representation (s.repr.)* is a pair $\langle S, B \rangle$, where the *constraint store* S is a multiset of constraint atoms and the *built-in store* B is a conjunction of

[3] Raiser et al [27] defined "state" similarly to what we call state representation, and they defined an operational semantics over equivalence classes of such states. We have taken the natural step of promoting such equivalence classes to be our states.

built-ins; any s.repr. with an unsatisfiable built-in store is considered identical to *failure*. Two s.repr.s $\langle S, B\rangle$ and $\langle S', B'\rangle$ are *variants* whenever, either[4]

- they are both *failure*, or
- there is a renaming substitution ρ such that $\mathcal{B} \models \forall(B\rho \rightarrow \exists(S\rho = S' \wedge B')) \wedge \mathcal{B} \models \forall(B' \rightarrow \exists(S\rho = S' \wedge B\rho))$

A *state* is an equivalence class of s.repr.s under the variant relationship. For simplicity of notation, we typically indicate a state by one of its s.repr.s.

A *rule application* w.r.t. to a non-failure state $\langle S, B\rangle$ is a pre-application $H_1\backslash H_2$ `<=>` $G \,|\, C$ for which $\mathcal{B} \models B \rightarrow \exists_L G$, where L is the list of its local variables. There are two sorts of transitions, *by rule application* and *by built-in.*

$$\langle H_1 \uplus H_2 \uplus S, B\rangle \mapsto_{logic} \langle H_1 \uplus C \uplus S, G \wedge B\rangle$$
when there exists a rule application $H_1\backslash H_2$`<=>`$G\,|\,C$,
$$\langle \{b\} \uplus S, B\rangle \mapsto_{logic} \langle S, b \wedge B\rangle \qquad \text{for a built-in } b.$$

5 Confluence under the logic-based semantics re-explained, and why invariants are difficult

Here we explain the results of [1,2], also summarized in [16], using abstract simulation. Object and meta level systems coincide and are given by a CHR program under the logic-based semantics. Two rules give rise to a critical corner if a state can be constructed in which one rule consumes constraints that the other one needs to be applied; in that case, rule applications do not commute and a specific proof of joinability must be considered. We anticipate the re-use of the construction, when invariants are introduced: in a *pre*-corner, the guards are not necessarily satisfied (but may be so in the context of an invariant).

Definition 4. *Consider two rules* $r\colon H_1\backslash H_2$ `<=>` $G\,|\,C$ *and* $r'\colon H_1'\backslash H_2'$ `<=>` $G'\,|\,C'$ *renamed apart, and let* A *and* A' *be non-empty sets of constraints such that* $A \subseteq H_2$*,* $A' \subseteq H_1' \uplus H_2'$ *and* $\mathcal{B} \models \exists(A{=}A')$*. In that case, let*

$$\begin{aligned}
\bar{H} &= (H_1 \uplus H_2 \uplus H_1' \uplus H_2') \setminus A\\
s_0 &= \langle \bar{H}, (G \wedge G' \wedge A{=}A')\rangle\\
s &= \langle \bar{H}\backslash H_2 \uplus C, (G \wedge G' \wedge A{=}A')\rangle\\
s' &= \langle \bar{H}\backslash H_2' \uplus C', (G \wedge G' \wedge A{=}A')\rangle
\end{aligned}$$

When $s \neq s'$*,* s_0 *is a* critical, common ancestor state*, and* $s \,{}_{logic}\!\!\leftarrow\!\!\!\shortmid\, s_0 \mapsto_{logic} s'$ *is a* critical α pre-corner*; the constraints* A *(or* A'*) is called the* overlap *of* r *and* r'*. When, furthermore,* $\mathcal{B} \models \exists(G \wedge G' \wedge A{=}A')$*, it is a* critical α *corner.*

[4] An equation between multisets should be understood as an equation between suitable permutations of their elements.

The simulation is given by the following cover function.

$$\begin{aligned}
[\![\langle S,B\rangle]\!]^{logic}_{logic} &= \{\langle S\uplus S^+, B\wedge B^+\rangle \mid \\
&\qquad S^+ \text{ is a multiset of user and built-in constraints,}\\
&\qquad B^+ \text{ is a conjunction of built-ins}\,\}\\
[\![\langle S,B\rangle \mapsto_{logic} \langle S',B'\rangle]\!]^{logic}_{logic} &= \{(\langle S\uplus S^+, B\wedge B^+\rangle \mapsto_{logic} \langle S'\uplus S^+, B'\wedge B^+\rangle) \mid\\
&\qquad S^+ \text{ is a multiset of user and built-in constraints,}\\
&\qquad B^+ \text{ is a conjunction of built-ins, } \exists(B\wedge B^+) \text{ holds}\}
\end{aligned}$$

It is easy to check that this definition satisfies the conditions for being an abstract simulation given in Section 3, relying on *monotonicity*: $\mathcal{B} \models B \wedge B^+ \to \exists_L G$.

It can be shown that any corner not covered by a critical corner (Definition 4) is trivially joinable, see the extended report [6]. Thus, according to Lemmas 1 and 3, the program under investigation is confluent whenever it is terminating and this set of critical corners is joinable. The set of critical corners is finite and that allows for automatic confluence proofs by checking the critical corners, one by one, e.g., [24].

Example 3. Consider the one-rule `set`-program of Example 1, ignoring invariant and state equivalence. There are two critical corners, given by the two ways, the rule can overlap with itself:

$$\begin{array}{cc}
\langle\{\texttt{set([X1|L])},\texttt{item(X2)}\},\mathit{true}\rangle & \langle\{\texttt{set([X|L1])},\texttt{set(L2)}\},\mathit{true}\rangle\\
\uparrow_{logic} & \uparrow_{logic}\\
\langle\{\texttt{item(X1)},\texttt{set(L)},\texttt{item(X2)}\},\mathit{true}\rangle & \langle\{\texttt{set(L1)},\texttt{item(X)},\texttt{set(L2)}\},\mathit{true}\rangle\\
\downarrow_{logic} & \downarrow_{logic}\\
\langle\{\texttt{item(X1)},\texttt{set([X2|L])}\},\mathit{true}\rangle & \langle\{\texttt{set(L1)},\texttt{set([X|L2])}\},\mathit{true}\rangle
\end{array}$$

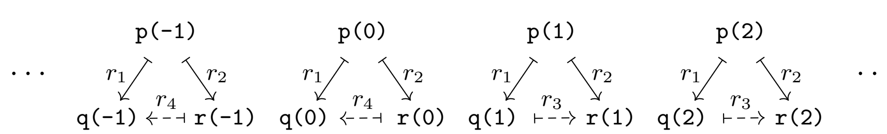
None of these corners are joinable, so the program is not confluent.

The simulation defined above, relying on monotonicity, do not generalize well for confluence under invariant, referred to as "observable confluence" in [13].

Example 4. Consider the CHR program consisting of the following four rules.

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

It is not confluent as its single critical corner `q(X)` $\leftarrow\!\dashv$ `p(X)` $\mapsto$ `r(X)` is not joinable (the built-in stores are *true* and thus omitted). However, adding the invariant "reachable from an initial state `p(`n`)` where n is an integer" makes it confluent. We indicate the set of all non-trivial object level corners as follows, with the dashed transitions proving each of them joinable.



These object 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) that covers all object corners. In other words, the smallest set of meta level corners that covers this set is the set itself. This was also noticed in [13] that used a construction that essentially reduces to the abstract simulation shown above.

The abstract simulation given by $[\![-]\!]^{logic}_{logic}$ of Definition 4 above defines an abstract interpretation, whose abstract domain is the complete lattice of CHR states ordered by the substate relationship (Section 3). Referring to Example 4, for instance the join of the infinite set of states $\{\langle \mathtt{p}(t), b\rangle \mid t$ is a term, b is a conjunction of built-ins$\}$ is $\langle \mathtt{p(X)}, \mathit{true}\rangle$. When the grounding invariant is introduced, the join operator is not complete; an attempt to join, say, $\langle \mathtt{p(0)}, \mathit{true}\rangle$ and $\langle \mathtt{p(1)}, \mathit{true}\rangle$ would not satisfy the invariant.[5]

6 Invariants and modulo equivalence

A program is typically developed with an intended set of queries in mind, giving rise to a state invariant, which may make an otherwise non-confluent program observably confluent (mod. equiv.). We can indicate a few general patterns of invariants and their possible effect on confluence.

- Elimination of non-joinable critical corners that do not cover any object corner satisfying the invariant. This was shown in Example 4 above, and is also demonstrated in the continuation of Example 3 (Ex. 7, below): "only one `set` constraint allowed".
- Making it possible to apply a given rule, which otherwise could not apply, e.g., providing a "missing" head constraint or enforcing guard satisfaction:
  1. "if a state contains `p(`*something*`)`, it also contains `q(`*the-same-something*`)`",
  2. "if a state contains `p(`*something*`)`, this *something* is a constant > 1".

An invariant of type 1 ensures confluence mod. equiv. of a version of the Viterbi algorithm [8]; an invariant of type 2 is indicated in Example 4 and formalized in Example 6, below.

As shown in Example 4 above, invariants block for a direct re-use CHR's logical semantics as its own meta-level and, accordingly, existing methods and confluence checkers. In some cases, it is possible to eliminate invariants by program transformations, so that rules apply exactly when the invariant and the original rule guards are satisfied; this means that the transformed program is confluent if and only if the original one is confluent under the invariant.

Example 5. Reconsidering the program of Example 4, the following is an example of such a transformed program; the constants `a` and `b` are introduced as representations of positive, resp., non-positive integers.

[5] Such an attempt might be $\langle \mathtt{p(X)}, (\mathtt{X{=}0} \vee \mathtt{X{=}1})\rangle$; notice that `X` is a variable, thus breaking the invariant.

```
p(a) <=> q(a).    p(a) <=> r(a).    p(a) <=> r(a).
p(b) <=> q(b).    p(b) <=> r(b).    r(b) <=> q(b).
```

Such program transformations become more complex when the guards describe more involved dependencies between the head variables. More importantly, invariants that exclude certain constraints in a state cannot be expressed in this way, for example "only one `set` constraint allowed" (Examples 3 and 7). Thus we refrain from pursuing a transformational approach. To obtain a maximum degree of generality, we introduce a meta level formalization of CHR's operational semantics that include representations as explicit data objects of states and their components, possibly parameterized by constrained meta variables.

6.1 The choice of a ground representation

Invariants and state equivalences are inherently meta level statements, as they are *about* states, and may refer to notions inexpressible at the object level, e.g., that some part being ground or a variable. Earlier work on meta-interpreters for logic programs, e.g., [5,18,19], offers the desired expressibility in terms of a *ground representation.* Any object term, formula, etc. is named by a ground meta level term. Variables are named by special constants, say `X` by `'X'`, and any other symbol by a function symbol written the same way; e.g., the non-ground object level atom `p(A)` is named by the ground meta level term `p('A')`. For any such ground meta level term mt, we indicate the object it names as $[\![mt]\!]^{Gr}$. For example, $[\![\mathtt{p('A')}]\!]^{Gr} = \mathtt{p(A)}$ and $[\![\mathtt{p('A')} \wedge \mathtt{'A'>2}]\!]^{Gr} = (\mathtt{p(A)} \wedge \mathtt{A>2})$.

For a given object entity e, we define its *lifting* to the meta level by 1) selecting a meta level term that names e, and 2) replacing variable names in it consistently by fresh meta level variables. For example, $\mathtt{p(X)} \wedge \mathtt{X>2}$ is lifted to $\mathtt{p(}x\mathtt{)} \wedge x\mathtt{>2}$, where `X` and x are object, resp., meta variables. By virtue of this overloaded syntax, we may read such an entity e (implicitly) as its lifting.

A collection of *meta level constraints* is assumed whose meanings are given by a theory $\mathcal{M}$. We start defining meta level states without detailed assumptions about $\mathcal{M}$, that are postponed to Definition 6 below. We assume object level built-in theory $\mathcal{B}$, invariant I_{logic} and state equivalence $\approx_{logic}$.

Definition 5. *A* constrained meta level term *is a structure of the form* $(mt \text{ WHERE } M)$*, where* mt *is a meta level term and* M *a conjunction of* $\mathcal{M}$ *constraints. We define*

$$[M] \quad = \quad \{\sigma \mid \mathcal{M} \models M\sigma\},$$
$$[\![mt \text{ WHERE } M]\!]^{meta}_{logic} \quad = \quad \{[\![mt\,\sigma]\!]^{Gr} \mid \sigma \in [M]\}.$$

A meta level state representation (s.repr.) *is a constrained meta level term* $st \text{ WHERE } M$ *for which* $[\![st \text{ WHERE } M]\!]^{meta}_{logic}$ *is a set of object level states. Two meta level s.repr.s* SR_1*,* SR_2 *are* variants *whenever each object level s.repr. in* $[\![SR_1]\!]^{meta}_{logic}$ *is a variant of some object level s.repr. in* $[\![SR_2]\!]^{meta}_{logic}$ *and vice versa. A* meta level state *is an equivalence class of meta level s.repr.s under the variant* *relationship. For structures of meta level states (transitions, corners, etc.), we apply the following convention, where f may represent any such structure.*

$$\begin{array}{ll} & [\![f(mt_1 \text{ WHERE } M_1, \ldots, mt_n \text{ WHERE } M_n)]\!]^{meta}_{logic} \\ = & [\![f(mt_1, \ldots, mt_n) \text{ WHERE } M_1 \wedge \ldots \wedge M_n]\!]^{meta}_{logic} \end{array}$$

Meta level invariant I^{meta}_{logic} and equivalence $\approx^{meta}_{logic}$ are defined as follows.

- $I^{meta}_{logic}(S)$ *whenever* $I_{logic}(s)$ *for all* $s \in [\![S]\!]^{meta}_{logic}$.
- $S_1 \approx^{meta}_{logic} S_2$ *whenever* $s_1 \approx_{logic} s_2$ *for all* $(s_1, s_2) \in [\![(S_1, S_2)]\!]^{meta}_{logic}$.

As before, we may indicate a meta level state by a representation of it.

Definition 6. *The theory $\mathcal{M}$ includes at least the following constraints.*

- `=`/2 *with its usual meaning of syntactic identity,*
- Type constraints `type`/2. *For example* `type(var,`x`)` *is true in $\mathcal{M}$ whenever x is the name of an object level variable;* `var` *is an example of a* type, *and we introduce more types below when we need them.*
- Modal constraints ☺F *and* ☹F *defined to be true in $\mathcal{M}$ whenever* $\mathcal{B} \models [\![F]\!]^{Gr}$*, resp.,* $\mathcal{B} \models [\![\neg F]\!]^{Gr}$.
- *We define two constraints* `inv` *and* `equiv` *such that* `inv`(Σ) *is true in $\mathcal{M}$ whenever* $[\![\Sigma]\!]^{Gr}$ *is an I_{logic} state (representation) of the logical semantics, and* `equiv`(Σ_1, Σ_2) *whenever* $[\![(\Sigma_1, \Sigma_2)]\!]^{Gr}$ *is a pair of states (representations) (s_1, s_2) of the logical semantics such that* $s_1 \approx_{logic} s_2$.
- `freshVars(`L`,`T`)` *is true in $\mathcal{M}$ whenever L is a list of all different variables names, none of which occur in the term T;* `freshVars(`L_1`,`L_2`,`T`)` *abbreviates* `freshVars(`L_{12}`,`T`))` *where L_{12} is the concat. of L_1 and L_2.*

Definitions 5 and 6 comprise the first steps towards a simulation of the logic-based semantics, and we continue with the last part, the transition relation.

Definition 7. *Consider a (lifted version of a) pre-application* $H_1 \backslash H_2$ `<=>` G`|`C *with local variables L and a consistent meta level state (S WHERE M) with* $S = \langle H_1 \uplus H_2 \uplus S^+, B^+\rangle$ *and*

$$\mathcal{M} \quad \models \quad M \;\to\; \big(\texttt{inv}(S) \wedge ☺B^+ \wedge ☺(B^+ \to \exists_L G) \wedge \texttt{freshVars(}L\texttt{,}S\texttt{)}\big).$$

Then the following is a meta level transition by rule application.

$$S \text{ WHERE } M \qquad \longmapsto^{meta}_{logic} \qquad \langle H_1 \uplus C \uplus S^+, G \wedge B^+\rangle \text{ WHERE } M$$

Consider a (lifted version of a) built-in b of $\mathcal{B}$ and a consistent meta level state (S WHERE M) with $S = \langle \{b\} \uplus S^+, B^+\rangle$ *and*

$$\mathcal{M} \quad \models \quad M \;\to\; \big(\texttt{inv}(S) \wedge ☺B^+\big).$$

Then the following is a meta level transition by built-in.

$$\langle \{b\} \uplus S^+, B^+\rangle \text{ WHERE } M \qquad \longmapsto^{meta}_{logic} \qquad \langle S^+, b \wedge B^+\rangle \text{ WHERE } M$$

Notice that for both sorts of transitions, the implication of $\smiley B^+$ excludes transitions from failed and mixed states. For built-in transitions, the resulting states may be non-failed, failed or mixed.

Lemma 4. *For a given CHR program with I_{logic} and $\approx_{logic}$, the definitions of meta level states and transitions $\longmapsto^{meta}_{logic}$, I^{meta}_{logic} and $\approx^{meta}_{logic}$, together with $[\![-]\!]^{meta}_{logic}$ comprise an abstract simulation of the logic-based semantics.*

Transitions are not possible from a mixed or failed meta level state, but modal constraints are useful for restricting to the relevant substate, such that transitions are known to exists. This is expressed by the following propositions that are immediate consequences of the definitions.

Proposition 1. *Let $r\colon H_1\backslash H_2$ `<=>` G`|`C be a (lifted version of a) pre-application with local variables L and $\Sigma = (\langle S,B\rangle$ WHERE $M)$ a meta level state with $H_1 \uplus H_2 \subseteq S$. Whenever the meta level state $\Sigma^{\smiley} = (\langle S,B\rangle$ WHERE $M \wedge \widehat{M})$ is consistent, with $\widehat{M} = \mathtt{inv}(\langle S,B\rangle) \wedge \smiley B \wedge \smiley(B \rightarrow \exists_L G) \wedge \mathtt{freshVars(}L\mathtt{,}\Sigma\mathtt{)}$, there exists a meta level rule application by r,*

$$\Sigma^{\smiley} \quad \longmapsto^{meta}_{logic} \quad \langle S\backslash H_2 \uplus C, B \wedge G\rangle \text{ WHERE } M \wedge \widehat{M}.$$

Furthermore, $\Sigma^{\smiley}$ is the greatest substate of Σ to which r can apply.

Proposition 2. *Let b be a (lifted version of a) built-in and $\Sigma = (\langle S,B\rangle$ WHERE $M)$ a meta level state with $b \in S$. When $\Sigma^{\smiley} = (\langle S,B\rangle$ WHERE $M \wedge \widehat{M})$ is consistent, with $\widehat{M^{\smiley}} = \mathtt{inv}(\langle S,B\rangle) \wedge \smiley B \wedge \smiley(B \rightarrow b)$, there is a meta level trans.,*

$$\Sigma^{\smiley} \quad \longmapsto^{meta}_{logic} \quad \langle S\backslash\{b\}, B \wedge b\rangle \text{ WHERE } M \wedge \widehat{M^{\smiley}}.$$

Whenever $\Sigma^{\frownie} = (\langle S,B\rangle$ WHERE $M \wedge \widehat{M^{\frownie}})$ is consistent, with $\widehat{M^{\frownie}} = \mathtt{inv}(\langle S,B\rangle) \wedge \smiley B \wedge \frownie(B \rightarrow b)$, there is a meta level transition by b,

$$\Sigma^{\frownie} \quad \longmapsto^{meta}_{logic} \quad \langle S\backslash\{b\}, B \wedge b\rangle \text{ WHERE } M \wedge \widehat{M^{\frownie}}.$$

The state $\Sigma^{\smiley}$ (resp. $\Sigma^{\frownie}$) is the greatest substate of Σ for which the meta level transition by b leads to a non-failure and non-mixed (resp. failed) state.

With Propositions 1 and 2 in mind, we define meta level critical corners from the critical corners of Definition 4.

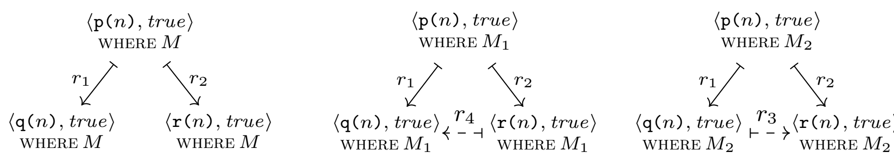
Definition 8. *Let $\langle S_1,B_1\rangle \leftarrowtail_{logic} \langle S_0,B_0\rangle \mapsto_{logic} \langle S_2,B_2\rangle$ be a (lifted version of a) critical α pre-corner given by Def. 4, in which the leftmost (rightmost) rule application has local variables L_1 (L_2) and guard G_1 (G_2). Assume S^+ and B^+ are fresh meta level variables and let, for $i = 0,1,2$,*

$$\begin{aligned}
\Sigma_i &= \langle S_i \uplus S^+, B_i \wedge B^+\rangle \\
M &= \mathtt{inv}(\Sigma_0) \wedge \smiley B_0 \wedge \smiley(B_0 \wedge B^+ \rightarrow \exists_{L_1} G_1) \wedge \smiley(B_0 \wedge B^+ \rightarrow \exists_{L_2} G_2) \wedge \\
&\quad \mathtt{freshVars(}L_1\mathtt{,}L_2\mathtt{,}\Sigma\mathtt{)}
\end{aligned}$$

When $(\Sigma_0$ WHERE $M)$ *is consistent, the following is* a critical meta level α corner*.*

$$(\Sigma_1 \text{ WHERE } M) \;\; \longleftarrow\!\!\!\dashv^{meta}_{logic} \;\; (\Sigma_0 \text{ WHERE } M) \;\; \longmapsto^{meta}_{logic} \;\; (\Sigma_2 \text{ WHERE } M)$$

Example 6. (Continuing Ex.4) The invariant is formalized at the meta level as states of the form $\langle\{\mathit{pred}(n)\}, \mathit{true}\rangle$ WHERE `type(int,`n`)` where *pred* is one of `p`, `q` and `r`. Below is shown the non-joinable critical meta level corner generated by Def. 8. It is split-joinable as demonstrated by its splitting into two corners; each shown joinable by the indicated dotted transition. Let M stand for the meta-level constraint `type(int,`n`)`, M_1 for $M \wedge \text{☺} n \leq 0$ and M_2 for $M \wedge \text{☺} n > 0$.



According to Lemma 5 shown below, the program is confluent.

When, furthermore, a state equivalence $\approx_{logic}$ is assumed, we need also show joinability of β corners, i.e., those composed by an equivalence and a transition.

Definition 9. *Let* $H \backslash H'$ `<=>` G`|`C *be a (lifted version of a) variant of a rule with local variables* L*. Assume* S^+*,* B^+ *and* Σ_1 *are fresh meta-variables, and let*

$$\Sigma_0 \;=\; \langle H \uplus H' \uplus S^+, B^+\rangle \qquad\qquad \Sigma_2 \;=\; \langle H \uplus C \uplus S^+, G \wedge B^+\rangle$$
$$M \;=\; \mathtt{inv}(\Sigma_0) \wedge \text{☺} B \wedge \text{☺}(B \rightarrow \exists L G) \wedge \mathtt{freshVars(}L\mathtt{,}\Sigma_0\mathtt{)} \wedge \mathtt{equiv}(\Sigma_0, \Sigma_1)$$

When $(\Sigma_0$ WHERE $M)$ *is consistent, the following is* a critical meta level β corner by rule application*.*

$$(\Sigma_1 \text{ WHERE } M) \approx^{meta}_{logic} (\Sigma_0 \text{ WHERE } M) \longmapsto^{meta}_{logic} (\Sigma_2 \text{ WHERE } M)$$

Let b *be a (lifted version of a) built-in atom whose arguments are fresh variables. Assume* S^+*,* B^+ *and* Σ_1 *are fresh meta-variables, and let*

$$\Sigma_0 \;=\; \langle\{b\} \uplus S^+, B^+\rangle \qquad\qquad \Sigma_2 \;=\; \langle S^+, b \wedge B^+\rangle$$
$$M \;=\; \mathtt{inv}(\Sigma_0) \wedge \text{☺} B \wedge \mathtt{freshVars(}L\mathtt{,}\Sigma_0\mathtt{)} \wedge \mathtt{equiv}(\Sigma_0, \Sigma_1)$$

When $(\Sigma_0$ WHERE $M)$ *is consistent, the following is* a critical meta level β corner by built-in*.*

$$(\Sigma_1 \text{ WHERE } M) \approx^{meta}_{logic} (\Sigma_0 \text{ WHERE } M) \longmapsto^{meta}_{logic} (\Sigma_2 \text{ WHERE } M)$$

Lemma 5. *Let a terminating CHR program* Π *with invariant* I_{logic} *(and state quivalence* $\approx_{logic}$*) be given. Then* Π *is confluent (modulo* $\approx_{logic}$*) if and only if its set of critical corners (Def.s 8–9) is split-joinable w.r.t.* I^{meta}_{logic} *(modulo* $\approx^{meta}_{logic}$*).*

Example 7 (Cont. Ex. 3; adapted from [8]). The invariant is formalized at the meta level as states of the form

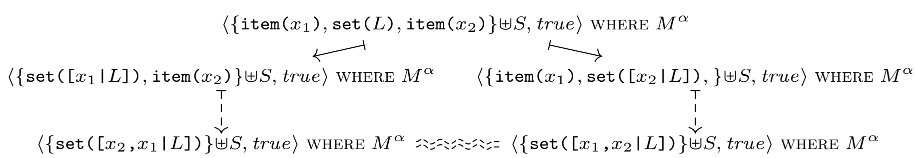
$$\langle\{\texttt{set(}L\texttt{)}\}\uplus S, true\rangle \text{ WHERE } \texttt{type(constList,}L\texttt{)}\wedge\texttt{type(constItems,}S\texttt{)};$$

we assume types `const` for all constants, `constList` for all lists of such, and `constItems` for sets of constraints of the form `item(`c`)` where c is a constant.

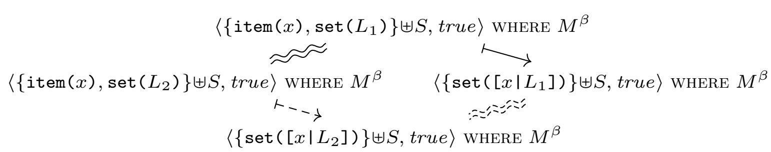
The state equivalence is formalized at the meta level as the relationships of states of the following form, where `perm(`L_1`,`L_2`)` means that L_1 and L_2 are lists being permutations of each other; and $M^{\approx}$ stands for `type(constList,`L_1`)`$\wedge$ `type(constList,`L_1`)`$\wedge$`perm(`L_1`,`L_2`)`$\wedge$`type(constItems,`S`)`,

$$\langle\{\texttt{set(}L_1\texttt{)}\}\uplus S, true\rangle \text{ WHERE } M^{\approx} \quad \approx^{meta}_{logic} \quad \langle\{\texttt{set(}L_2\texttt{)}\}\uplus S, true\rangle \text{ WHERE } M^{\approx}$$

The critical object level corner with two set constraints in the states does not give rise to a critical meta level corner as the invariant is not satisfied. The other one is shown here, including (with dotted arrows) its proof of joinability modulo equivalence; M^{α} stands for `type(const,`x_1`)` $\wedge$ `type(constList,`L`)` $\wedge$ `type(const,`x_2`)` $\wedge$ `type(constItems,`S`)`.



We consider the following critical meta level β corner. M^{β} stands for `type(const,`x`)` $\wedge$`type(constList,`L_1`)`$\wedge$`type(constList,`L_2`)`$\wedge$`perm(`L_1`,`L_2`)`$\wedge$`type(constItems,`S`)`.



All critical corners are joinable modulo equivalence, and since the program is obviously terminating, Lemma 5 gives that the program is confluent mod. equiv.

7 Conclusion

We generalized the critical pair approach using a meta level simulation to prove confluence under invariant and modulo equivalence for Constraint Handling Rules. We have demonstrated how this principle makes it possible to express natural invariants and equivalences, that cannot be expressed in CHR itself, in a formal way at the meta level, anticipating machine supported proofs using a meta level constraint solver, based on a ground representation. A constraint solver is currently under development, partly inspired by [5]. Depending on the complexity of the invariants and equivalences – and of the CHR programs under investigation – it may be difficult to obtain a complete solver.

For simplicity of notation, we did not include mechanisms to prevent loops caused by propagation rules; [8] has included this in a meta level representation for the Prolog based semantics, and is easily adapted for the logic based semantics exposed in the present paper. For comparison with earlier work on confluence for CHR, we used here a logic-based CHR semantics, which has nice theoretical properties, but is incompatible with standard implementations of CHR and applies only for a limited set of programs. In [9], we have defined meta level constraints and a simulation for an alternative CHR semantics [7,8] that reflects CHR's Prolog based implementation, including a correct handling of Prolog's non-logical devices (e.g., `var`/1, `nonvar`/2, `is`/2) and runtime errors.

The abstract simulations used for the classical CHR confluence results are special cases of abstract interpretations, but when invariants are introduced – or when considering full CHR including Prolog-style non-logical devices, cf. [9] – this correspondence does not hold. The concept of abstract simulations and their use for proving confluence (mod. equiv.) seem obvious to investigate for a large variety of rewrite based systems, e.g., constrained term rewriting, conditional term rewriting, interactive theorem provers, and rule-based specifications of abstract algorithms.

Acknowledgement

We thank the anonymous reviewers for their insightful comments, suggesting to compare with a transformational approach, cf. Example 5, and helping us to clarify the relationship between abstract simulation and abstract interpretation.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *CP97*, volume 1330 of *LNCS*, pages 252–266. Springer, 1997.
2. S. Abdennadher, T. W. Frühwirth, and H. Meuss. On confluence of constraint handling rules. In *CP96*, 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. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.
5. H. Christiansen. Automated reasoning with a constraint-based metainterpreter. *Journal of Logic Programming*, 37(1-3):213–254, 1998.
6. H. Christiansen and M. Kirkeby. *Confluence of CHR revisited: invariants and modulo equivalence. [Extended version with proofs]*, volume 153 of *Computer Science Research Report*. Roskilde University, October 2018. `https://forskning.ruc.dk/files/63000759/ChrKir_LOPSTR2018_ExtReport.pdf`.
7. H. Christiansen and M. H. Kirkeby. Confluence modulo equivalence in Constraint Handling Rules. In *LOPSTR*, volume 8981 of *LNCS*, pages 41–58. Springer, 2014.
8. H. Christiansen and M. H. Kirkeby. On proving confluence modulo equivalence for constraint handling rules. *Formal Aspects of Computing*, 29(1):57–95, 2017.

9. H. Christiansen and M. H. Kirkeby. Towards a constraint solver for proving confluence with invariant and equivalence of realistic CHR programs. In *WFLP*, volume 11285 of *LNCS*, 2018. To appear.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, pages 238–252, 1977.
11. P. Curien and G. Ghelli. On confluence for weakly normalizing systems. In *RTA-91*, volume 488 of *LNCS*, pages 215–225, 1991.
12. 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.
13. G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable confluence for Constraint Handling Rules. In *ICLP 2007*, volume 4670 of *LNCS*, pages 224–239. Springer, 2007.
14. T. W. Frühwirth. User-defined constraint handling. In *ICLP 1993*, pages 837–838. MIT Press, 1993.
15. T. W. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
16. T. W. Frühwirth. *Constraint Handling Rules*. Cambridge Uni. Press, 2009.
17. D. Gall and T. W. Frühwirth. Confluence modulo equivalence with invariants in Constraint Handling Rules. In *Functional and Logic Programming – FLOPS, Proceedings*, volume 10818 of *LNCS*, pages 116–131. Springer, 2018.
18. P. Hill and J. Gallagher. Meta-programming in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 421–497. Oxford Science Publications, Oxford University Press, 1994.
19. P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In *Meta-programming in logic programming*, pages 23–51. The MIT Press, 1988.
20. C. Holzbaur and T. W. Frühwirth. A PROLOG Constraint Handling Rules compiler and runtime system. *Applied Artificial Intelligence*, 14(4):369–388, 2000.
21. G. P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
22. J. Jaffar and J. Lassez. Constraint logic programming. In *Symposium on Principles of Programming Languages. POPL 1987*, pages 111–119. ACM Press, 1987.
23. M. H. Kirkeby and H. Christiansen. Confluence and convergence modulo equivalence in probabilistically terminating reduction systems. *International Journal of Approximate Reasoning*, 105:217–228, 2019.
24. 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.
25. M. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.
26. D. M. R. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science, 5th GI-Conference, Proceedings*, pages 167–183, 1981.
27. F. Raiser, H. Betz, and T. W. Frühwirth. Equivalence of CHR states revisited. In *Proc. Int'l Workshop on Constraint Handling Rules*, Report CW 555, pages 33–48. Katholieke Universiteit Leuven, Belgium, 2009.
28. T. Schrijvers and B. Demoen. The K.U.Leuven CHR System: Implementation and Application. In *Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5. Ulmer Informatik-Berichte, Nr. 2004-01, 2004.