Observable Confluence for Constraint Handling Rules

Gregory J. Duck[1], Peter J. Stuckey[1], and Martin Sulzmann[2]

[1] NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, 3010, AUSTRALIA
{gjd,pjs}@cs.mu.oz.au
[2] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Abstract. Constraint Handling Rules (CHR) are a powerful rule based language for specifying constraint solvers. Critical for any rule based language is the notion of confluence, and for terminating CHR programs there is a decidable test for confluence. But many CHR programs that are in practice confluent fail this confluence test. The problem is that the states that illustrate non-confluence are not observable from the initial goals of interest. In this paper we introduce the notion of observable confluence, a more general notion of confluence which takes into account whether states are observable. We devise a test for observable confluence which allows us to verify observable confluence for a range of CHR programs dealing with agents, type systems, and the union-find algorithm.

1 Introduction

Constraint Handling Rules [3] (CHR) are a powerful rule based language for specifying constraint solvers. Constraint handling rules operate on a global multi-set (conjunction) of constraints. A constraint handling rule defines a rewriting from one multi-set of constraints to another.

A critical issue for any rule based language is the notion of confluence. A CHR program is confluent if all possible rewriting sequences from a given input lead to the same result. Thus confluent programs have a "deterministic behaviour" with respect to the input goals, i.e. given some input goal, we can uniquely determine the output. For terminating CHR programs there is a decidable test for confluence [1]. Unfortunately there are many (terminating) programs which are confluent in practice, but fail to pass the test.

In this paper, we make the following contributions:

- We introduce the notion of *observable confluence* which generalises the notion of confluence by only considering rewriting steps which are observable with respect to some invariant (Section 3.3).
- We give a generalised confluence test where we only need to consider joinability of critical pairs satisfying the invariant (Section 4).

– We show that the generalised confluence test enables us to verify observable confluence of CHR used for the specification of agents, the union-find algorithm, and type systems (Section 5). All of these classes of CHR programs are non-confluent under the standard notion.

To the best of our knowledge, we are the first to study observable confluence in the context of a rule-based language. In the workshop papers [2, 5], we reported some preliminary results. The present work represents a significantly revised and extended version of [2].

We continue in Section 2 where we consider a number of motivating examples. Section 3 provides background material on CHR.

2 Motivating Examples

The following examples fail the standard confluence test, but we can show that they are observable confluent with respect to some appropriate invariant.

2.1 Blocks World

In our first example, we consider a set of CHRs used for agent-oriented programming [5]. The following CHR program fragment describes the behaviour of an agent in a blocks world:[3]

```
g1 @ get(X), empty   <=> hold(X).
g2 @ get(X), hold(Y) <=> hold(X), clear(Y).
```

The constraint `hold(X)` denotes that the agent holds some element `X`, whereas `empty` denotes that the agent holds nothing. The constraint `clear(Y)` simply represents the fact that `Y` is not held. The constraint `get(X)` represents an action, to get some element `X`. The atoms preceding the '`@`' symbols are the *rule names*, thus the rules are named `g1` and `g2` respectively. Both rules are *simplification rules*, rewriting constraints matching the left-hand-side to the constraints in the right-hand-side. The first rule rewrites the constraints `get(X)` ∧ `empty` to `hold(X)`. The second rule rewrites `get(X)` ∧ `hold(Y)` to `hold(X)` ∧ `clear(Y)`.

It is clear that the rules are *non-confluent*. Consider the *critical state* `get(X)` ∧ `hold(Y)` ∧ `empty` formed by combining the heads of rules `g1` and `g2`. This critical state can be rewritten to either `hold(Y)` ∧ `hold(X)` by applying rule `g1`, or to `hold(X)` ∧ `clear(Y)` ∧ `empty` by applying rule `g2`. These two derived states are a *critical pair* between the two rules. The confluence test for CHR [1] states that a terminating program is confluent iff all critical pairs are *joinable*, i.e. can be rewritten to the same result. Since no rewriting steps can join `hold(Y)` ∧ `hold(X)` and `hold(X)` ∧ `clear(Y)` ∧ `empty`, the blocks world program is non-confluent.

Let us consider the non-confluent state `get(X)` ∧ `hold(Y)` ∧ `empty` more closely: it represents the agent holding nothing (`empty`) whilst simultaneously holding an object Y (`hold(Y)`). Clearly, such a state is nonsense, so we would

[3] CHR follows a Prolog like notation, where identifiers starting with a lower case letter indicate predicates and function symbols, and identifiers starting with upper case letters indicate variables.

like to exclude it from consideration. To do this we need a weaker notion of confluence, i.e. confluence with respect to *valid* states. We refer to this as *observable confluence*.

Specifically, we can informally define valid states as follows:

> "Either the agent holds some element `X` or holds nothing."
> "There is at most one `get(_)` operation at one time."

The conjunction of these conditions is an invariant in the blocks world program. In this paper, we show that all non-confluent states violate this invariant, thus blocks world program is observable confluent.

A similar form of observable confluence under some invariant arises in our next example.

2.2 Union Find

Consider the following program which is part[4] of the simple union-find code from [9, 8].

```
union    @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).
findNode @ X ~> PX \ find(X,R) <=> find(PX, R).
findRoot @ root(X) \ find(X,R) <=> R = X.
linkEq   @ link(X,X) <=> true.
link     @ link(X,Y), root(X), root(Y) <=> Y ~> X, root(X).
```

Both `findNode` and `findRoot` are *simpagation rules*, which are similar to simplification rules, except constraints on the LHS of the '\' symbol are not rewritten. This program defines an environment where the `root(X)` and `X ~> Y` constraints define trees, and `union(X,Y)` links trees so that they have the same root.

The union-find program is non-confluent, since there are eight non-joinable critical pairs. The authors of [8] classify the critical pairs as either *avoidable* (as in they should not arise in practice) and *unavoidable* (as inherent non-confluence in the union-find algorithm). For example, the critical state between the `linkEq` and `link` is `link(X,X)` $\wedge$ `root(X)` $\wedge$ `root(X)`, with two `root(X)` constraints.[5] The critical pair is `root(X)` $\wedge$ `root(X)` and `X ~> X` $\wedge$ `root(X)`, which is non-joinable. However, in [8] it is argued that this critical pair is *avoidable*, since the presence of two `root(X)` in the state violates the definition of a tree (i.e. X can only be the root of one tree). This kind of reasoning can be understood in terms of invariants and observable confluence.

As was the case with the blocks world example, we define an invariant that describes valid states. Firstly, we informally define *validTrees*, as follows:

> "For all `X` there is at most one `root(X)` or `X ~> Y`, and there are no cycles `X ~>` $\mathtt{Y}_1$, ..., $\mathtt{Y}_n$ `~> X`"

We are also interested in the confluence of a single `union(X,Y)` operation executed in isolation on valid trees. Some combinations of operations are not valid, thus we define *validOps* as follows:

[4] We have removed the `make` rule to simplify the invariant.
[5] CHR uses a multi-set semantics, thus here we consider $X \wedge X$ to be distinct from X.

> "There is at most one `union(_,_)` or `link(_,_)`, and if there is a `union(_,_)` there is no `find(_,_)`."

This condition helps enforce confluence: since the order in which these operations are executed can affect the final result. For example, executing a `find` before or after a `union` may produce different results, since the `union` may update the trees.

To ensure observable confluence, there is one final case to consider: a concurrent `link` and `find` operation. In this case, we can not simply make these operations mutually exclusive, since `link` and `find` do interact in the body of the `union` rule. However, the second argument to a `find` constraint must always be associated to the corresponding `link` constraint. Therefore, we define $validFind$ as follows:

> "If there is a `find(X,Y)` then there is a `link(Y,_)` or `link(_,Y)` but no `link(Y,Y)`, and `Y` does not appear in any other constraint."

Define $\mathcal{U} = validTrees \wedge validOps \wedge validFind$, then we verify that $\mathcal{U}$ is preserved by rule application, and thus is an invariant. Furthermore, none of the non-joinable critical pairs, or any states extended from these critical pairs, satisfy $\mathcal{U}$.[6] Therefore the union-find program P is observable confluent with respect to $\mathcal{U}$. Or in other words, P is $\mathcal{U}$-confluent.

This shows that the `union` operation, executed in isolation on valid trees, is confluent, even though the program itself is not confluent.

2.3 Type Class Functional Dependencies

The invariants we have seen so far reject (critical) states by observing the constraints in the store. But in CHR, sometimes a state cannot be observable because of the *kind* of rules that have, or have not, fired so far. This is due to CHR *propagation rules* which only add new constraints but do not delete existing constraints. To avoid trivial non-termination, the CHR semantics maintains a *propagation history* to avoid re-application of the same rule on the same set of constraints. The short story is that certain states can never be observable because of their propagation histories. Our next example illustrates this point.

We consider CHRs which arise from the translation of type class constraints in Haskell [7] involving functional dependencies [4]. We directly give the CHRs and omit the type class program.

```
r1 @ f(int,bool,float)    <=> true.
r2 @ f(A,B1,C), f(A,B2,D) ==> B1 = B2.
r3 @ f(int,B,C)           ==> B = bool.
```

The first rule is a simplification rule. The second and third rule are *propagation rules*. Propagation rules do not delete the constraints matching the head, thus they only add constraints. For example, the second rule adds the constraint

[6] In the original paper [8], one of the critical pairs, namely `link` + `findRoot`, was "unavoidable" under the author's conditions. However, we have sufficiently strengthened the conditions to make the "unavoidable" critical pair avoidable.

`B1 = B2` whenever we see `f(A,B1,C)` $\wedge$ `f(A,B2,D)` in the store. To avoid trivial non-termination, propagation rules maintain a history of applications, and avoid applying the same propagation rule more than once on the same set of constraints.

When testing for confluence, we examine critical states, which are minimal states where two different rule firings are possible. In order to be truly minimal, the propagation history is assumed to be as strong as possible, i.e. where no propagation rule can fire, except possibly the two rules used to generate the critical state itself.

Rules `r1`, `r2` give rise to the critical state `f(int,bool,float)`$\wedge$`f(int,B2,D)` from which we can derive two different states as shown by the following rewriting steps `f(int,bool,float)` $\wedge$ `f(int,B2,D)` $\rightarrowtail_{r1}$ `f(int,B2,D)` and

$$\begin{aligned} & \texttt{f(int,bool,float)} \wedge \texttt{f(int,B2,D)} \\ \rightarrowtail_{r2}\ & \texttt{f(int,bool,float)} \wedge \texttt{f(int,bool,D)} \wedge \texttt{B2 = bool} \\ \rightarrowtail_{r1}\ & \texttt{f(int,bool,D)} \wedge \texttt{B2 = bool} \end{aligned}$$

Note that we cannot apply the rule `r3` to the state `f(int,B2,D)` because the propagation history in the critical state disallows this. As the critical state has led to two different non-joinable states, the above CHR program is non-confluent.

But in practice the critical state `f(int,bool,float)` $\wedge$ `f(int,B2,D)` where the propagation history prevents rule `r3` from firing on the second constraint cannot arise. The initial state always begins with an empty (weakest) propagation history. Hence, rule `r3` must have fired already on the second constraint. If this were the case, then the constraint $B2 = bool$ should appear in the critical state, but $B2 = bool$ does not occur. Therefore, the critical state is not *reachable* from any initial goal. Further details are given in Section 5.1, where we show that this program is in fact observable confluent with respect to the reachability invariant.

Next, we review background material on CHR before introducing the notion of observable confluence and the observable confluence test.

3 Preliminaries

A CHR *simpagation* rule is of the form $(r \ @\ H_1' \setminus H_2' \iff g \mid C)$ where we propagate H_1' and simplify H_2' by C if the guard g is satisfied. We call r a *propagation* rule if H_2' is empty and a *simplification* rule if H_1' empty. As seen in Section 2, $(r \ @\ H_2' \iff g \mid C)$ is shorthand for the simplification rule $(r \ @\ \emptyset \setminus H_2' \iff g \mid C)$, and $(r \ @\ H_1' \Longrightarrow g \mid C)$ is shorthand for the propagation rule $(r \ @\ H_1' \setminus \emptyset \iff g \mid C)$.

In CHR there are two distinct types of constraints: *user constraints* and *built-in constraints*. Built-in constraints are provided by an external solver, whereas user constraints are defined by the rules. Only user constraints may appear in the rule head (H_1' and H_2'), and only built-in constraints in the guard g. The body C may contain both kinds of constraints.

Formally, CHR is a reduction system $\langle \rightarrowtail, \Sigma \rangle$ where $\rightarrowtail$ is the CHR rewrite relation and Σ is the set of all *CHR states*.

Definition 1 (CHR State). *A* state *is a tuple of the form*

$$\langle G, S, B, \mathcal{T}, \mathcal{V} \rangle$$

where goal G is a multi-set of constraints (both user and built-in), user store S is a multi-set of user constraints, built-in store B is a conjunction of built-in constraints, token store $\mathcal{T}$ is a set of tokens*, and variables of interest $\mathcal{V}$ is the set of variables present in the initial goal. Throughout this paper we use symbol 'σ' to represent a state, and Σ to represent the set of all states.* □

The *built-in constraint store* B contains any built-in constraint that has been passed to the built-in solver. Since we will usually have no information about the internal representation of B, we treat it as a conjunction of constraints. We assume $\mathcal{D}$ denotes the theory for the built-in constraints B.

The *token store*[7] $\mathcal{T}$ is a set of *tokens* of the from $(r@C)$, where r is a rule name, and C is a sequence of user constraints. A CHR propagation rule r may only be applied to C if the token $(r@C)$ exists in the token store. This is necessary to prevent trivial non-termination for propagation rules. Whenever a new constraint is added to the user store, the *token set* of that constraint is added to the token store.

Definition 2 (Token Set). *Let P be a CHR program, C be a set of user constraints, and S a user-store, then define*

$$T_{(C,S)} = \{r@H' \mid (r@H \Longrightarrow g \mid B) \in P, H' \subseteq C \uplus S, C \subseteq H', H' \text{ unifies with } H\}$$

to be the token set *of C with respect to S.* □

In the above, we write $\uplus$ for multi-set union. Later, we will also use multi-set intersection $\uplus$.

We define $vars(o)$ as the free variables in some object o: e.g. term, formula, constraint. We define an *initial state* as follows.

Definition 3 (Initial State). *Given a multi-set of constraints G (i.e. the* goal*) the* initial state *with respect to G is $\langle G, \emptyset, true, \emptyset, vars(G)\rangle$.* □

The operational semantics of CHR[8] is based on the following three transitions which map states to states:

Definition 4 (Operational Semantics).
1. Solve $\langle \{c\} \uplus G, S, B, \mathcal{T}, \mathcal{V}\rangle \rightarrowtail \langle G, S, c \wedge B, \mathcal{T}, \mathcal{V}\rangle$
where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, \mathcal{T}, \mathcal{V}\rangle \rightarrowtail \langle G, \{c\} \uplus S, B, T_{(\{c\},S)} \uplus \mathcal{T}, \mathcal{V}\rangle$
where c is a user constraint.
3. Apply $\langle G, H_1 \uplus H_2 \uplus S, B, T \uplus \mathcal{T}, \mathcal{V}\rangle \rightarrowtail \langle C \uplus G, H_1 \uplus S, \theta \wedge B, \mathcal{T}, \mathcal{V}\rangle$
where there exists a (renamed apart) rule $(r \ @ \ H_1' \setminus H_2' \iff g \mid C)$ in P, and $T = \{(r@H_1, H_2)\}$ if $H_2' = \emptyset$, otherwise $T = \emptyset$. The matching substitution θ is such that

$$\begin{cases} H_1 = \theta(H_1') \\ H_2 = \theta(H_2') \\ \mathcal{D} \models B \rightarrow \exists \bar{a}(\theta \wedge g) \end{cases}$$

[7] The *token store* is also known as the *propagation history*.

[8] There are many different versions of the operational semantics of CHR. In this paper we use a version that is close to the original operational semantics described in [1]. This version is the most suitable for the study of confluence.

where $\bar{a} = vars(g) - vars(H'_1, H'_2)$ *and* $\mathcal{D}$ *denotes the built-in theory.* □

A *derivation* is a sequence of states connected by transitions. We use notation $\sigma_0 \rightarrowtail^* \sigma_1$ to represent a derivation from σ_0 to σ_1.

3.1 Confluence

Confluence depends on the notion of equivalence between CHR states. The equivalence relation for CHR states is known as *variance*:

Definition 5 (Variance). *Two states*

$$\sigma_1 = \langle G_1, S_1, B_1, \mathcal{T}_1, \mathcal{V} \rangle \qquad \textit{and} \qquad \sigma_2 = \langle G_2, S_2, B_2, \mathcal{T}_2, \mathcal{V} \rangle$$

are variants *(written* $\sigma_1 \approx \sigma_2$*) if there exists a unifier* ρ *of* S_1 *and* S_2*,* G_1 *and* G_2*,* $(\mathcal{T}_1 \Cap T_{(S_1,\emptyset)})$ *and* $(\mathcal{T}_2 \Cap T_{(S_2,\emptyset)})$ *such that*

1. $\mathcal{D} \models \bar{\exists}_{\mathcal{V}_1} B_1 \rightarrow \bar{\exists}_{\mathcal{V}_1} \rho \wedge B_2$
2. $\mathcal{D} \models \bar{\exists}_{\mathcal{V}_2} B_2 \rightarrow \bar{\exists}_{\mathcal{V}_2} \rho \wedge B_1$

where $\mathcal{V}_1 = \mathcal{V} \cup vars(G_1) \cup vars(S_1) \cup vars(\mathcal{T}_1)$ *and* $\mathcal{V}_2 = \mathcal{V} \cup vars(G_2) \cup vars(S_2) \cup vars(\mathcal{T}_2)$*. Otherwise the two states are variants if* $\mathcal{D} \models \neg \bar{\exists}_{\emptyset} B_1$ *and* $\mathcal{D} \models \neg \bar{\exists}_{\emptyset} B_2$ *(i.e. both states are* $false$*).* □

Confluence relies on whether two states can derive the same state. This property is known as *joinability*.

Definition 6 (Joinable). *Two states* σ_1 *and* σ_2 *are* joinable *if there exists states* σ'_1 *and* σ'_2 *such that* $\sigma_1 \rightarrowtail^* \sigma'_1$ *and* $\sigma_2 \rightarrowtail^* \sigma'_2$ *and* $\sigma'_1 \approx \sigma'_2$*. We use the notation* $(\sigma_1 \downarrow \sigma_2)$ *to indicate that* σ_1 *and* σ_2 *are joinable.* □

Finally, we can define confluence as follows:

Definition 7 (Confluence). *A CHR program* P *is* confluent *if the following holds for all states* σ_0*,* σ_1 *and* σ_2*: If* $\sigma_0 \rightarrowtail^* \sigma_1$ *and* $\sigma_0 \rightarrowtail^* \sigma_2$ *then* σ_1 *and* σ_2 *are joinable.* □

3.2 Confluence Test

In [1] it was shown that confluence is decidable for terminating CHR programs. The confluence test for CHR depends on calculating all critical pairs between rules in the program. First we define the notion of a *critical ancestor state*.

Definition 8 (Critical Ancestor States). *Given two (renamed apart) rule instances:* $(r1 \ @ \ H_1 \backslash H_2 \Longleftrightarrow g_1 \mid B_1)$ *and* $(r2 \ @ \ H_3 \backslash H_4 \Longleftrightarrow g_2 \mid B_2)$*, then the set of all* critical ancestor states *(or simply* ancestor states*)* $\Sigma_{\mathcal{CP}}$ *between* $r1$ *and* $r2$ *is:*

$$\left\{ \langle \emptyset, H^{\Delta}_{r1} \uplus H^{\Delta}_{r2} \uplus H^{\cap}_{r1}, H^{\cap}_{r1} = H^{\cap}_{r2} \wedge g_1 \wedge g_2, \mathcal{T}_{\mathcal{CP}}, \mathcal{V}_{\mathcal{CP}} \rangle \;\middle|\; \begin{array}{l} H_{r1} = H_1 \uplus H_2 \\ H_{r2} = H_3 \uplus H_4 \\ H_{r1} = H^{\cap}_{r1} \uplus H^{\Delta}_{r1} \\ H_{r2} = H^{\cap}_{r2} \uplus H^{\Delta}_{r2} \\ \mathcal{V}_{\mathcal{CP}} = \\ \quad vars(H_1 \wedge H_2 \wedge H_3 \wedge H_4 \wedge g_1 \wedge g_2) \end{array} \right\}$$

where, given $e_1 = (r1@H_1, H_2)$ *and* $e_2 = (r2@H_3, H_4)$*, then* $\mathcal{T}_{\mathcal{CP}} = \{e_i \mid i \in \{1, 2\}, r_i \textit{is a propagation rule}\}$.

Basically, a critical ancestor state is a minimal state applicable to both rules. The sets $H^{\cap}_{r1}$ and $H^{\cap}_{r2}$ represent some potential overlap between the two rules. If the rules heads H_{r1} and H_{r2} do not overlap, then $H^{\cap}_{r1} = H^{\cap}_{r2} = \emptyset$ gives the only non-$false$ ancestor state.

We define a *critical pair* in terms of an ancestor state.

Definition 9 (Critical Pair). *Given the rules $r1$, $r2$ and the set $\Sigma_{\mathcal{CP}}$ from Definition 8, for $\sigma_{\mathcal{CP}} \in \Sigma_{\mathcal{CP}}$ where*

$$\sigma_{\mathcal{CP}} = \langle \emptyset, H^{\Delta}_{r1} \uplus H^{\Delta}_{r2} \uplus H^{\cap}_{r1}, H^{\cap}_{r1} = H^{\cap}_{r2} \wedge g_1 \wedge g_2, \mathcal{T}_{\mathcal{CP}}, \mathcal{V}_{\mathcal{CP}} \rangle$$

then the critical pair (σ_A, σ_B) for $\sigma_{\mathcal{CP}}$ is

$$\begin{aligned}(&\langle B_1, (H^{\Delta}_{r1} \uplus H^{\Delta}_{r2} \uplus H^{\cap}_{r1}) - H_2, H^{\cap}_{r1} = H^{\cap}_{r2} \wedge g_1 \wedge g_2, \mathcal{T}_A, \mathcal{V}_{\mathcal{CP}} \rangle,\\ &\langle B_2, (H^{\Delta}_{r1} \uplus H^{\Delta}_{r2} \uplus H^{\cap}_{r1}) - H_4, H^{\cap}_{r1} = H^{\cap}_{r2} \wedge g_1 \wedge g_2, \mathcal{T}_B, \mathcal{V}_{\mathcal{CP}} \rangle)\end{aligned}$$

where $\mathcal{T}_A = \mathcal{T}_{\mathcal{CP}} - \{e_1\}$ and $\mathcal{T}_B = \mathcal{T}_{\mathcal{CP}} - \{e_2\}$ and where e_1 and e_2 are defined as in Definition 8. □

Informally, Definition 9 simply states that (σ_A, σ_B) is the result of respectively firing $r1$ and $r2$ on $\sigma_{\mathcal{CP}}$, whilst being careful to specify exactly *how* the rules were applied (e.g. how the constraints were matched against the rule head).

For the rest of the paper, we use $\sigma_{\mathcal{CP}}$ to denote the ancestor state of a critical pair $\mathcal{CP}$.

Confluence can be proven by showing that all critical pairs are joinable.

Theorem 1 (Confluence Test). *[1] Given a terminating CHR program P, if all critical pairs between all rules in P are joinable, then P is confluent.*

This is known as the *confluence test* for terminating CHR programs.

3.3 $\mathcal{I}$-Confluence

In this section we formally define $\mathcal{I}$-confluence (i.e. observable confluence)[9] with respect to an invariant $\mathcal{I}$.

Definition 10 (Invariant). *An* invariant $\mathcal{I}(\sigma)$ *is a property such that for all σ_0 and σ_1, we have that if $\sigma_0 \rightarrowtail \sigma_1$ (or $\sigma_0 \approx \sigma_1$) and $\mathcal{I}(\sigma_0)$ then $\mathcal{I}(\sigma_1)$.* □

Example 1 (Blocks World Invariant). First we define $\texttt{exists}(\sigma, M)$, which decides if the multi-set of user constraints M exists in σ:

$$\begin{aligned}&\texttt{exists}(\langle G,S,B,\mathcal{T},\mathcal{V}\rangle, M) \Leftrightarrow\\ &\qquad \exists S' \subseteq user(G) \uplus S \quad \wedge \quad \mathcal{D} \models builtin(G) \wedge B \rightarrow \exists_{vars(M)}(M = S')\end{aligned}$$

where $user(G)$ and $builtin(G)$ returns all user/built-in constraints in G respectively.

[9] The terminology "$\mathcal{I}$-confluence" and "observable confluence" are largely interchangeable. The latter is useful when referring to a specific invariant $\mathcal{I}$.

The invariant for the blocks world example from Section 2.1 is formally represented as $\mathcal{B}(\sigma)$ where

$$\begin{aligned}\mathcal{B}(\sigma) \Leftrightarrow & \neg\texttt{exists}(\sigma, \{\texttt{empty}, \texttt{empty}\}) \wedge \neg\texttt{exists}(\sigma, \{\texttt{empty}, \texttt{holds(\_)}\}) \wedge \\ & \neg\texttt{exists}(\sigma, \{\texttt{holds(\_)}, \texttt{holds(\_)}\}) \wedge \neg\texttt{exists}(\sigma, \{\texttt{get(\_)}, \texttt{get(\_)}\})\end{aligned}$$

The first three conditions state that the agent either holds something or holds nothing. The outcome is determined by the order in which `get` operations are executed. Therefore, we impose the fourth condition which guarantees that we only consider isolated `get` operations. It is straightforward to verify that the Blocks-world program maintains $\mathcal{B}$ as an invariant. □

Given an invariant $\mathcal{I}$, we define confluence with respect to $\mathcal{I}$ as follows:

Definition 11 (Observable Confluence). *A CHR program P is* $\mathcal{I}$-confluent *with respect to invariant $\mathcal{I}$ if the following holds for all states σ_0, σ_1 and σ_2 where $\mathcal{I}(\sigma_0)$ holds: If $\sigma_0 \rightarrowtail^* \sigma_1$ and $\sigma_0 \rightarrowtail^* \sigma_2$ then σ_1 and σ_2 are joinable.* □

Alternatively, a CHR program P is *$\mathcal{I}$-confluent* with respect to invariant $\mathcal{I}$ iff the reduction system $\mathcal{R} = \langle \{\sigma \in \Sigma | \mathcal{I}(\sigma)\}, \rightarrowtail \rangle$ is confluent. Likewise, P is *$\mathcal{I}$-local-confluent* iff $\mathcal{R}$ is local-confluent and P is *$\mathcal{I}$-terminating* iff $\mathcal{R}$ is terminating.

Observable confluence is a weaker form of confluence,[10] thus the standard confluence test (see Theorem 1) is too strong. We desire a more general test for observable confluence.

4 Observable Confluence

4.1 Extensions

To help reduce the level of verbosity, we introduce the notion of a state *extension*.

Definition 12 (Extension). *A state $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V} \rangle$ can be* extended *by another state $\sigma_e = \langle G_e, S_e, B_e, \mathcal{T}_e, \mathcal{V}_e \rangle$ as follows*

$$\sigma \oplus \sigma_e = \langle G \uplus G_e, S \uplus S_e, B \wedge B_e, \mathcal{T} \uplus \mathcal{T}_e, \mathcal{V}_e \rangle$$

We say that σ_e is an extension *of σ.* □

An *extension* σ_e adds some "extra" information to an existing state σ. Notice that the variables of interest $\mathcal{V}$ in the original state σ are simply replaced by variables of interest $\mathcal{V}_e$ from state σ_e. We also assume that $\approx$ (see Definition 5) is the equivalence relation for extensions.

Example 2. The following equations are of the form $\sigma \oplus \sigma_e = \sigma'$ where σ and σ' are states, and σ_e is an extension.

$$\begin{aligned}\langle \emptyset, \{p(X)\}, true, \emptyset, \emptyset \rangle \oplus \langle \emptyset, \{q(X)\}, true, \emptyset, \emptyset \rangle &= \langle \emptyset, \{p(X), q(X)\}, true, \emptyset, \emptyset \rangle \\ \langle \emptyset, \{p(X)\}, true, \emptyset, \emptyset \rangle \oplus \langle \emptyset, \emptyset, X = 0, \emptyset, \emptyset \rangle &= \langle \emptyset, \{p(X)\}, X = 0, \emptyset, \emptyset \rangle \\ \langle \emptyset, \{p(X)\}, true, \emptyset, \emptyset \rangle \oplus \langle \emptyset, \emptyset, true, \emptyset, \{X\} \rangle &= \langle \emptyset, \{p(X)\}, true, \emptyset, \{X\} \rangle\end{aligned}$$

The first adds a user constraint $q(X)$ to the user store, the second adds a built-in constraint $X = 0$ to the built-in store, and the third replaces the variables of interest with the set $\{X\}$. □

[10] Observable confluence is only strictly weaker if $\mathcal{I} \neq true$.

One crucial property of extensions is that they do not affect the applicability of the CHR rewrite relation $\rightarrowtail$.

Lemma 1. *For all states σ and σ_1 such that $\sigma \rightarrowtail^* \sigma_1$, and for all extensions σ_e have that $\sigma \oplus \sigma_e \rightarrowtail^* \sigma_1 \oplus \sigma_e$.*

The notions of *variance* and *joinability* depend on the variables of interest $\mathcal{V}$. Therefore we must refine the definition of extension to ensure joinability is preserved.

Definition 13 (Valid Extension). *A* valid extension $\sigma_e = \langle G_e, S_e, B_e, \mathcal{T}_e, \mathcal{V}_e \rangle$ *of a state $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V} \rangle$ is an extension such that*

$$v \in vars(G, S, B, \mathcal{T}) \wedge v \notin \mathcal{V} \Rightarrow v \notin vars(G_e, S_e, B_e, \mathcal{T}_e, \mathcal{V}_e)$$

Example 3. Consider the state $\sigma = \langle \emptyset, \{leq(X,Y)\}, X = Y, \emptyset, \{X\} \rangle$. Then $\sigma_e = \langle \{leq(X,Z)\}, \emptyset, true, \emptyset, \{X\} \rangle$ is a valid extension of σ. However, the extension $\sigma'_e = \langle \{leq(Y,Z)\}, \emptyset, true, \emptyset, \{X\} \rangle$ is invalid since local variable Y is mentioned in the extension. □

For valid extensions, joinability is preserved.

Lemma 2. *For all states $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V} \rangle$, σ_1, and σ_2 such that*

$$\sigma \rightarrowtail^* \sigma_1 \quad \textit{and} \quad \sigma \rightarrowtail^* \sigma_2$$

If $\sigma_1 \downarrow \sigma_2$, then for all valid extensions σ_e we have that $\sigma_1 \oplus \sigma_e \downarrow \sigma_2 \oplus \sigma_e$.

4.2 $\mathcal{I}$-Confluence

If all variables in a state $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V} \rangle$ are in $\mathcal{V}$, i.e. $vars(G, S, B, \mathcal{T}) \subseteq \mathcal{V}$, then all extensions are valid for σ. The ancestor state of a critical pair has this property, thus proving $\mathcal{I}$-confluence is equivalent to showing that for all critical pairs (σ_A, σ_B) with ancestor state $\sigma_{\mathcal{CP}}$, and all extensions σ_e such that $\mathcal{I}(\sigma_{\mathcal{CP}} \oplus \sigma_e)$ holds, then $(\sigma_A \oplus \sigma_e, \sigma_B \oplus \sigma_e)$ are joinable. The problem is that the set of all extensions is infinite, so we need some way of reducing the number of extensions that we must test.

Our strategy is to define a partial order[11] $\preceq_\sigma$ over valid extensions that satisfy the invariant with respect to some state σ.

Definition 14 (Partial Order). *Given a state $\sigma = \langle G, S, B, \mathcal{T}, \mathcal{V} \rangle$, and valid extensions σ_{e1} and σ_{e2} of σ, then we define $\sigma_{e1} \preceq_\sigma \sigma_{e2}$ to hold if*

1. *there exists a valid extension σ_{e3} of $(\sigma \oplus \sigma_{e1})$ such that $(\sigma \oplus \sigma_{e1}) \oplus \sigma_{e3} \approx \sigma \oplus \sigma_{e2}$*
2. *$\mathcal{V} - \mathcal{V}_{e1} \subseteq \mathcal{V} - \mathcal{V}_{e2}$ holds.* □

We find that if $\sigma_{e1} \preceq_\sigma \sigma_{e2}$, $\sigma \rightarrowtail \sigma_1$, and $\sigma \rightarrowtail \sigma_2$, then $(\sigma_1 \oplus \sigma_{e1} \downarrow \sigma_2 \oplus \sigma_{e1})$ implies $(\sigma_1 \oplus \sigma_{e2} \downarrow \sigma_2 \oplus \sigma_{e2})$. The means that the $\preceq_\sigma$ order respects joinability, and thus reduces the number of states that must be tested in order to prove confluence.

We define the following for notational convenience.

[11] Although we believe that relation $\preceq_\sigma$ is a partial order, we omit a formal discussion since none of our theoretical results require it to be.

Definition 15. *Let $\Sigma_e(\sigma)$ be the set of all valid extensions of some state σ, and let $\Sigma_e^{\mathcal{I}}(\sigma) = \{\sigma_e | \sigma_e \in \Sigma_e(\sigma) \wedge \mathcal{I}(\sigma \oplus \sigma_e)\}$ be the set of all valid extensions satisfying the invariant $\mathcal{I}$. Finally, let $\mathcal{M}_e^{\mathcal{I}}(\sigma)$ be the $\prec_\sigma$-minimal elements of $\Sigma_e^{\mathcal{I}}(\sigma)$.* □

We define the following property, which we show to be equivalent to $\mathcal{I}$-local-confluence.

Definition 16. *A program P is* minimal extension joinable *if for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$ with ancestor state $\sigma_{\mathcal{CP}}$, and for all $\sigma_e \in \mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$, we have that $(\sigma_1 \oplus \sigma_e, \sigma_2 \oplus \sigma_e)$ is joinable.*

Lemma 3. *Given that $\prec_{\sigma_{\mathcal{CP}}}$ is well-founded for all critical pairs $\mathcal{CP}$, then: P is $\mathcal{I}$-local-confluent iff P is minimal extension joinable.*

For terminating programs, we invoke Newman's Lemma [6] to show that $\mathcal{I}$-local-confluence implies $\mathcal{I}$-confluence.

Theorem 2. *For all $\mathcal{I}$-terminating programs P, given that $\prec_{\sigma_{\mathcal{CP}}}$ is well-founded for all critical pairs $\mathcal{CP}$, then: P is $\mathcal{I}$-confluent iff P is minimal extension joinable.*

4.3 $\mathcal{I}$-Confluence Test

The standard confluence test for terminating CHR programs relies on showing that all critical pairs are joinable. Based on Theorem 2, we can define a similar test for the more general notion of $\mathcal{I}$-confluence. Instead of testing critical pairs, we test critical pairs $\mathcal{CP}$ extended by the $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ extension set.

For Theorem 2 to be used in practice, there are two issues that must be resolved: (1) the order $\prec_{\sigma_{\mathcal{CP}}}$ must be *well-founded*, and (2) for each critical pair $\mathcal{CP}$, the set of extensions $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ must be *computable*.

Well-foundedness. Ordering $\prec_{\sigma_{\mathcal{CP}}}$ is essentially a product order over the fields in the CHR state. Thus for the G, S, $\mathcal{T}$ fields of a state, $\prec_{\sigma_{\mathcal{CP}}}$ is the well-founded subset ordering with the minimal element $G = S = \mathcal{T} = \emptyset$. The set of variables of interest $\mathcal{V}$ is ordered differently. In this case, extensions are ordered based on the *difference* between $\mathcal{V}$ and some given reference set $\mathcal{V}_0$. Again, this is (a variant of) subset ordering with the minimal element $\mathcal{V} = \mathcal{V}_0$.

Where well-foundedness may be broken is the built-in store B. Indeed, for some constraint domains, the set of extensions is not well-founded.

Example 4. Consider the constraint domain $\mathcal{D}$ of (in)equalities over the integers. Consider the following sequence of extensions: $\sigma_e^i = \langle \emptyset, \emptyset, X < i, \emptyset, \{X\} \rangle$. Since for all j, k such that $k > j$ we have that $\mathcal{D} \models X < j \leftrightarrow (X < k \wedge X < j)$ we have that $\sigma_e^j \prec_\sigma \sigma_e^k$ holds. Since the sequence is infinite, the relation $\prec_\sigma$ is not well-founded. □

There are also important examples of constraint domains that do preserve well-foundedness:

Proposition 1. *The order $\prec_\sigma$ is well-founded for all σ if $\mathcal{D}$ is equations over the Herbrand domain.*

Proposition 2. *The order $\prec_\sigma$ is well-founded for all σ if $\mathcal{D}$ is a finite domain.*

We can use Proposition 2 to find a practical solution to Example 4. Instead of considering all possible integers, we can restrict ourselves to some finite range of integers (e.g. those representable on a 32-bit CPU). The example is now well founded, with the minimal element $\langle \emptyset, \emptyset, X < 2^{32}, \emptyset, \{X\} \rangle$.

Computability. Depending on the invariant $\mathcal{I}$, the set $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ may be either *undecidable* or be *infinite*. Even if $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ is decidable and finite, an algorithm to compute it is dependent on the nature of $\mathcal{I}$. The computation of $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ is therefore instance-dependent.

Despite this, in Section 5 we look at several instances for $\mathcal{I}$ and compute the $\mathcal{M}_e^{\mathcal{I}}(\sigma_{\mathcal{CP}})$ for each critical pair.

5 Examples

We use Theorem 2 to verify observable confluence under the invariants we have seen earlier in Section 2. In addition, we verify ground confluence.

5.1 Reachable Confluence

A naive definition of confluence states that: a program P is confluent if for all input I, there is only one possible O such that $I \rightarrowtail^* O$. However, in [2] it was shown that there exist non-confluent CHR programs that satisfy this alternative definition. In this section, we reformulate the main theorem from [2] in terms of our observable confluence results.

The key issue is the difference between *reachable* and *unreachable* states. A *reachable* state is one that can be derived from some initial state (i.e. from some initial goal).

Definition 17 (Reachability). *We define the property that a state is* reachable $R(\sigma)$ *as follows:*

- *For all initial states $\sigma_i = \langle G, \emptyset, true, \emptyset, vars(G) \rangle$ $R(\sigma_i)$ holds; and*
- *If $\sigma_1 \rightarrowtail \sigma_2$ (or $\sigma_1 \approx \sigma_2$) and $R(\sigma_1)$ holds, then $R(\sigma_2)$ holds.* □

By definition, $R(\sigma)$ is an invariant.

The naive definition of confluence is more precisely defined as R-confluence, i.e. confluence with respect to the reachability invariant. In some systems, e.g. in term rewriting, all states (terms) are potential initial states, and thus R-confluence and confluence are equivalent. However, as was show in Section 2.3, the same is not true for CHR. Our main counter-example is the following class of CHR programs, which arise from the study of multi-parameter typeclasses with functional dependencies [10].

Definition 18 (FD-CHR). *A CHR program P is said to be in the* FD-CHR *class of programs if it is of the form*

`r1 @ p(`$X_1, ..., X_d, X_{d+1}, ..., X_r, ...$`), p(`$X_1, ..., X_d, Y_{d+1}, ..., Y_r, ...$`) ==>`
$X_{d+1} = Y_{d+1}, ..., X_r = Y_r$.
`r2 @ p(`$f_1, ..., f_n$`) <=>` B .
`r3 @ p(`$f_1, ..., f_d, Y_1, ..., Y_r, ...$`) ==>` $Y_1 = f_{d+1}, ..., Y_r = f_r$.

where B is an arbitrary conjunction of built-in and user constraints, and f_i are arbitrary terms such that $vars(f_{d+1}, ..., f_r) \subseteq vars(f_1, ..., f_d)$ We also require P to be terminating. Here the indices $1..d$ represent the domain *and indices $(d+1)..r$ represent the* range *of the functional dependency. Also note that r is allowed to be less than n.* □

In [2] it was shown that the *FD-CHR* class of programs are R-confluent, however many instances of Definition 18 are not confluent.

Example 5 (FD-CHR). Consider the following instance of Definition 18:[12]

```
r1 @ f(A,B1,C), f(A,B2,D) ==> B1 = B2.
r2 @ f(int,bool,float)    <=> true.
r3 @ f(int,B,C)           ==> B = bool.
```

Consider the critical pair (σ_1, σ_2) between rules `r1` and `r2`:

$$\sigma_{\mathcal{CP}} = \langle \emptyset, \{f(int, bool, float), f(int, B2, D)\}, true, \{t\}, \{B2, D\}\rangle$$
$$\sigma_1 = \langle \{bool = B2\}, \{f(int, bool, float), f(int, B2, D)\}, true, \emptyset, \{B2, D\}\rangle$$
$$\sigma_2 = \langle \emptyset, \{f(int, B2, D)\}, true, \{t\}, \{B2, D\}\rangle$$

where t is the token $(r1@f(int, bool, float), f(int, B2, D))$. The final states derived from σ_1 and σ_2 are:

$$\sigma_1 \rightarrowtail^* \langle \emptyset, \{f(int, bool, D)\}, B2 = bool, \emptyset, \{B2, D\}\rangle$$
$$\sigma_2 \rightarrowtail^* \langle \emptyset, \{f(int, B2, D)\}, true, \{t\}, \{B2, D\}\rangle$$

These states are not variants. In the final state for σ_1, the variable $B2$ is constrained to *bool*, but this is not the case for the final state for σ_2. Thus the critical pair is not joinable, and the program is not confluent. □

State $\sigma_{\mathcal{CP}}$ is not reachable, since the lack of a token $(r3@f(int, B2, D))$ suggests rule `r3` has already fired on constraint $f(int, B2, D)$. If that rule did fire, then we would expect the built-in store to entail $B2 = bool$, which is not the case.

Thus, we consider the minimal set of extensions that make $\sigma_{\mathcal{CP}}$ reachable. This set is:

$$\mathcal{M}_e^R(\sigma_{\mathcal{CP}}) = \{\langle \emptyset, \emptyset, true, \{(r3@f(int, B2, D))\}, \mathcal{V}\rangle, \langle \{B2 = bool\}, \emptyset, true, \emptyset, \mathcal{V}\rangle,$$
$$\langle \emptyset, \emptyset, B2 = bool, \emptyset, \mathcal{V}\rangle\}$$

It is easy to verify that for all $\sigma_e \in \mathcal{M}_e^R(\sigma_{\mathcal{CP}})$ we have that $\sigma_1 \oplus \sigma_e \downarrow \sigma_2 \oplus \sigma_e$. We can verify similar results for all other critical pairs in P, and thus, by Theorem 2, program P is R-confluent.

We can generalise this basic approach, and restate the main theorem from [2].

[12] An informal version of this example was seen in Section 2.3.

Corollary 1. *All programs $P \in$FD-CHR are R-confluent.*

The alternative proof for Corollary 1 in [2] relied on showing that all programs $P \in$*FD-CHR* were related to a class of confluent programs, and that the relation was sufficient to show R-confluence. In this paper, the proof relies on Theorem 2, and thus is a far more direct proof of R-confluence.

5.2 Simple Confluence

It is common for programmers to implement non-confluent CHR programs that are well behaved for some certain input. For example, the union-find program [8] (also see Section 2.2) is non-confluent, however it is well behaved provided the initial goal satisfies some certain conditions.

Let $\mathcal{I}$ be an invariant that simply excludes non-joinable critical pairs from consideration, then P is always $\mathcal{I}$-confluent. We define this as *simple confluence*.

Corollary 2 (Simple Confluence). *Given an invariant $\mathcal{I}$ and an $\mathcal{I}$-terminating program P such that $\prec_\sigma$ is well-founded for all σ, then P is $\mathcal{I}$-confluent if for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$, either:*

1. *$\mathcal{I}(\sigma_{\mathcal{CP}})$ holds, and $\sigma_1 \downarrow \sigma_2$; or*
2. *For all extensions σ_e we have that $\mathcal{I}(\sigma_{\mathcal{CP}} \oplus \sigma_e)$ does not hold.*

Via the above corollary and the blocks world invariant $\mathcal{B}$ from Example 1, we can straightforwardly verify $\mathcal{B}$-confluence of the blocks world program from Section 2.1. Similarly, we can verify observable confluence of the union-find algorithm in Section 2.2.

5.3 Ground Confluence

A state σ is *ground*, i.e. $\mathcal{G}(\sigma)$ holds, if all variables $vars(\sigma)$ are constrained to be one value by the built-in store B of σ. Groundness is an invariant for *range restricted*[13] CHR programs. Typically, the critical pair between two rules is not ground, however we can invoke Theorem 2 to show $\mathcal{G}$-confluence.

Corollary 3 (Ground Confluence). *Given a $\mathcal{G}$-terminating, range restricted program P such that $\prec_\sigma$ is well-founded for all σ, then P is $\mathcal{G}$-confluent if for all critical pairs $\mathcal{CP} = (\sigma_1, \sigma_2)$ we have that $(\sigma_1 \oplus \sigma_e) \downarrow (\sigma_2 \oplus \sigma_e)$ for all extensions $\sigma_e \in \mathcal{M}(\sigma_{\mathcal{CP}})$ where:*

$$\mathcal{M}(\sigma_{\mathcal{CP}}) = \{\langle \emptyset, \emptyset, X_0 = d_0 \wedge ... \wedge X_n = d_m, \emptyset, \mathcal{V}_{\mathcal{CP}} \rangle \mid \{X_0, ..., X_n\} = vars(\sigma_{\mathcal{CP}}), d_i \in \mathcal{D}\}$$

If $\mathcal{D}$ is an finite set, then $\mathcal{M}(\sigma_{\mathcal{CP}})$ can be computed.

Example 6. Consider the following CHR program over the Boolean domain.

```
p(X,Y) <=> not(X,Y).          xor(0,0,Z) <=> Z = 0.
p(X,Y) <=> xor(1,X,Y).        xor(0,1,Z) <=> Z = 1.
not(0,Y) <=> Y = 1.           xor(1,0,Z) <=> Z = 1.
not(1,Y) <=> Y = 0.           xor(1,1,Z) <=> Z = 0.
```

[13] A CHR program is *range restricted* if $vars(H_1 \backslash H_2 \iff G|B) = vars(H_1 \wedge H_2)$ for all rules.

The critical state $\sigma_{\mathcal{CP}} = \langle \emptyset, \{p(X,Y)\}, true, \emptyset, \{X,Y\}\rangle$ between the first two rules is non-joinable, hence the program is non-confluent. Clearly $\mathcal{G}(\sigma_{\mathcal{CP}})$ does not hold, thus we evaluate $\mathcal{M}(\sigma_{\mathcal{CP}})$:

$$\begin{aligned}\mathcal{M}(\sigma_{\mathcal{CP}}) = \{&\langle \emptyset, \emptyset, X = 0 \wedge Y = 0, \emptyset, \{X,Y\}\rangle, \langle \emptyset, \emptyset, X = 0 \wedge Y = 1, \emptyset, \{X,Y\}\rangle, \\ &\langle \emptyset, \emptyset, X = 1 \wedge Y = 0, \emptyset, \{X,Y\}\rangle, \langle \emptyset, \emptyset, X = 1 \wedge Y = 1, \emptyset, \{X,Y\}\rangle\}\end{aligned}$$

For each of these extensions, the critical pair is joinable, and thus P is $\mathcal{G}$-confluent. □

6 Conclusion

We have shown that many non-confluent CHR programs are in fact observably confluent in practice, and have presented a method for proving the observable confluence of programs with respect to invariants. Furthermore, we have specialised our results for some common cases, such as simple confluence and ground confluence.

To the best of our knowledge, we are the first to study observable confluence in the context of a rule-based language. However, the notion of observable confluence could easily be extended to other areas, such as term rewriting, which is something we intend to investigate in the future.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.
2. G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable Confluence for Constraint Handling Rules. Technical Report CW 452, Katholieke Universteit Leuven, 2006. Proc. of CHR 2006, Third Workshop on Constraint Handling Rules.
3. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.
4. M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.
5. E. S. L. Lam and M. Sulzmann. Towards agent programming in CHR. Technical Report CW 452, Katholieke Universteit Leuven, 2006. Proc. of CHR 2006, Third Workshop on Constraint Handling Rules.
6. M. H. A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.
7. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
8. Tom Schrijvers and Thom W. Frühwirth. Analysing the CHR Implementation of Union-Find. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 135–146. Universität Ulm, Germany, 2005.
9. Tom Schrijvers and Thom W. Frühwirth. Optimal union-find in Constraint Handling Rules. *TPLP*, 6(1-2):213–224, 2006.
10. M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding Functional Dependencies via Constraint Handling Rules. *Journal of Functional Programming*, 17(1):83–129, 2007.