The Refined Operational Semantics of Constraint Handling Rules

Gregory J. Duck[1], Peter J. Stuckey[1], María García de la Banda[2], and Christian Holzbaur[3]

[1] Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{gjd,pjs}@cs.mu.oz.au
[2] School of Computer Science and Software Engineering
Monash University, Vic. 3800, Australia
mbanda@csse.monash.edu.au
[3] holzbaur@chello.at

Abstract. Constraint Handling Rules (CHRs) are a high-level rule-based programming language commonly used to write constraint solvers. The theoretical operational semantics for CHRs is highly non-deterministic and relies on writing confluent programs to have a meaningful behaviour. Implementations of CHRs use an operational semantics which is considerably finer than the theoretical operational semantics, but is still non-deterministic (from the user's perspective). This paper formally defines this *refined* operational semantics and proves it implements the theoretical operational semantics. It also shows how to create a (partial) confluence checker capable of detecting programs which are confluent under this semantics, but not under the theoretical operational semantics. This supports the use of new idioms in CHR programs.

1 Introduction

Constraint Handling Rules (CHRs) are a high-level rule-based programming language commonly used to write constraint solvers. The theoretical operational semantics of CHRs is relatively high level with several choices, such as the order in which transitions are applied, left open. Therefore, only confluent CHR programs, where every possible execution results in the same result, have a guaranteed behaviour.

This paper looks at the *refined* operational semantics, a more specific operational semantics which has been implicitly described in [10, 11], and is used by every Prolog implementation of CHRs we know of. Although some choices are still left open in the refined operational semantics, both the order in which transitions are applied and the order in which occurrences are visited, is decided. Unsurprisingly, the decisions follow Prolog style and maximise efficiency of execution. The remaining choices, which matching partner constraints are tried first, and the order of evaluation of CHR constraints awoken by changes in variables they involve, are left as choices for two reasons. First it is very difficult to see how a CHR programmer will be able to understand a fixed strategy in these cases. And second implementing a fixed strategy will restrict the implementation to be less efficient, for example by disallowing hashing index structures.

It is clear that CHR programmers take the refined operational semantics into account when programming. For example, some of the standard CHR examples are non-terminating under the theoretical operational semantics.

Example 1. Consider the following simple program that calculates the greatest common divisor (*gcd*) between two integers using Euclid's algorithm:

```
gcd1 @ gcd(0)           <=> true.
gcd2 @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

Rule `gcd1` is a simplification rule. It states that a fact `gcd(0)` in the store can be replaced by true. Rule `gcd2` is a simpagation rule, it states that if there are two facts in the store `gcd(`n`)` and `gcd(`m`)` where $m \geq n$, we can replace the part after the slash `gcd(`m`)` by the right hand side `gcd(`$m-n$`)`. [4] The idea of this program is to reduce an initial store of `gcd(A), gcd(B)` to a single constraint `gcd(C)` where `C` will be the *gcd* of `A` and `B`.

This program, which appears on the CHR webpage [6], is non-terminating under the theoretical operational semantics. Consider the constraint store `gcd(3), gcd(0)`. If the first rule fires, we are left with `gcd(3)` and the program terminates. If, instead, the second rule fires (which is perfectly possible in the theoretical semantics), `gcd(3)` will be replaced with `gcd(3-0) = gcd(3)`, thus essentially leaving the constraint store unchanged. If the second rule is applied indefinitely (assuming unfair rule application), we obtain an infinite loop.

In the above example, trivial non-termination can be avoided by using a *fair* rule application (i.e. one in which every rule that could fire, eventually does). Indeed, the theoretical operational semantics given in [7] explicitly states that rule application should be fair. Interestingly, although the refined operational semantics is not fair (it uses rule ordering to determine rule application), its unfairness ensures termination in the `gcd` example above. Of course, it could also have worked against it, since swapping the order of the rules would lead to non-termination.

The refined operational semantics allows us to use more programming idioms, since we can now treat the constraint store as a queryable data structure.

Example 2. Consider a CHR implementation of a simple database:

```
l1 @ entry(Key,Val) \ lookup(Key,ValOut) <=> ValOut = Val.
l2 @ lookup(_,_) <=> fail.
```

where the constraint `lookup` represents the basic database operations of key lookup, and `entry` represents a piece of data currently in the database (an entry in the database). Rule `l1` looks for the matching entry to a lookup query and returns in `ValOut` the stored value. Rule `l2` causes a lookup to fail if there is no matching entry. Clearly the rules are non-confluent in the theoretical operational semantics, since they rely on rule ordering to give the intended behaviour.

[4] Unlike Prolog, we assume the expression "$m - n$" is automatically evaluated.

The refined operational semantics also allows us to create more efficient programs and/or have a better idea regarding their time complexity.

Example 3. Consider the following implementation of Fibonacci numbers, `fib(N,F)`, which holds if F is the N^{th} Fibonacci number:

```
f1 @ fib(N,F) <=> 1 >= N | F = 1.
f2 @ fib(N,F0) \ fib(N,F) <=> N >= 2 | F = F0.
f3 @ fib(N,F) ==> N >= 2 | fib(N-2, F1), fib(N-1,F2), F = F1 + F2.
```

Rule `f3` is a propagation rule (as indicated by the `==>` arrow), which is similar to a simplification rule except the matching constraint `fib(`n`,`f`)` is not removed from the store.

The program is confluent in the theoretical operational semantics which, as we will see later, means it is also confluent in the refined operational semantics. Under the refined operational semantics it has linear complexity, while swapping rules `f2` and `f3` leads to exponential complexity. Since in the theoretical operational semantics both versions are equivalent, complexity is at best exponential.

We believe that Constraint Handling Rules under the refined operational semantics provide a powerful and elegant language suitable for general purpose computing. However, to make use of this language, authors need support to ensure their code is confluent within this context. In order to do this, we first provide a formal definition of the refined operational semantics of CHRs as implemented in logic programming systems. We then provide theoretical results linking the refined and theoretical operational semantics. Essentially, these results ensure that if a program is confluent under the theoretical semantics, it is also confluent under the refined semantics. Then, we provide a practical (partial) confluence test capable of detecting CHR programs which are confluent for the refined operational semantics, even though they are not confluent for the theoretical operational semantics. Finally, we study two CHR programs and argue our test is sufficient for real world CHR programs.

2 The Theoretical Operational Semantics ω_t

We begin by defining constraints, rules and CHR programs. For our purposes, a *constraint* is simply defined as an atom $p(t_1, ..., t_n)$ where p is some predicate symbol of arity $n \geq 0$ and $(t_1, ..., t_n)$ is an n-tuple of terms. A *term* is defined as either a variable X, or as $f(t_1, ..., t_n)$ where f is a function symbol of arity n and $t_1, ..., t_n$ are terms. Let $vars(A)$ return the variables occurring in any syntactic object A. We use $\bar{\exists}_A F$ to denote the formula $\exists X_1 \cdots \exists X_n F$ where $\{X_1, \ldots X_n\} = vars(F) - vars(A)$. We use $\bar{s} = \bar{t}$, where $\bar{s}$ and $\bar{t}$ are sequences, to denote the conjunction $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$.

Constraints can be divided into either CHR constraints or *builtin* constraints in some constraint domain $\mathcal{D}$. While the former are manipulated by the CHR execution algorithm, the latter are handled by an *underlying* constraint solver. Decisions about rule matchings will rely on the underlying solver proving that the current constraint store for the underlying solver entails a *guard* (a conjunction of builtin constraints). We will assume the solver supports (at least) equality.

There are three types of rules: simplification, propagation and simpagation. For simplicity, we consider both simplification and propagation rules as special cases of a simpagation rules. The general form of a *simpagation* rule is:

$$r \ @ \ H_1 \setminus H_2 \iff g \mid B$$

where r is the rule name, H_1 and H_2 are sequences of CHR constraints, g is a sequence of builtin constraints, and B is a sequence of constraints. If H_1 is empty, then the rule is a *simplification* rule. If H_2 is empty, then the rule is a *propagation* rule. At least one of H_1 and H_2 must be non-empty. Finally, a CHR program P is a sequence of rules.

We use $[H|T]$ to denote the first (H) and remaining elements (T) of a sequence, $++$ for sequence concatenation, ϵ for empty sequences, and $\uplus$ for multiset union. We shall sometimes treat multisets as sequences, in which case we nondeterministically choose an order for the objects in the multiset.

Given a CHR program P, we will be interested in numbering the occurrences of each CHR constraint predicate p appearing in the head of the rule. We number the occurrences following the top-down rule order and right-to-left constraint order. The latter is aimed at ordering first the constraints after the backslash (\) and then those before it, since this gives the refined operational semantics a clearer behaviour.

Example 4. The following shows the `gcd` CHR program of Example 1, written using simpagation rules and with all its occurrences numbered:

$\texttt{gcd1 @ } \epsilon \qquad\quad \texttt{\textbackslash\ gcd(0)}_1 \texttt{ <=> true} \quad \texttt{| true.}$
$\texttt{gcd2 @ gcd(N)}_3 \texttt{ \textbackslash\ gcd(M)}_2 \texttt{ <=> M} \geq \texttt{N | gcd(M-N).}$

2.1 The ω_t Semantics

Several versions of the theoretical operational semantics have already appeared in the literature, e.g. [1, 7], essentially as a multiset rewriting semantics. This section presents our variation, which is equivalent to previous ones, but is close enough to our refined operational semantics to make proofs simple.

Firstly, we define an *execution state*, as the tuple $\langle G, S, B, T\rangle_n$ where each element is as follows. The *goal* G is the multiset (repeats are allowed) of constraints to be executed. The CHR constraint *store* S is the multiset of *identified* CHR constraints that can be matched with rules in the program P. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i. This number serves to differentiate among copies of the same constraint. We introduce functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner.

The *builtin constraint store* B contains any builtin constraint that has been passed to the underlying solver. Since we will usually have no information about the internal representation of B, we will model it as an abstract logical conjunction of constraints. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints which fired a rule, and the name of the rule itself. This is necessary to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used to fire the rule before. Finally, the counter n represents the next free integer which can be used to number a CHR constraint.

Given an initial goal G, the *initial state* is: $\langle G, \emptyset, true, \emptyset\rangle_1$. The theoretical operational semantics ω_t is based on the following three transitions which map execution states to execution states:

1. Solve $\langle\{c\} \uplus G, S, B, T\rangle_n \rightarrowtail \langle G, S, c \wedge B, T\rangle_n$ where c is a builtin constraint.
2. Introduce $\langle\{c\} \uplus G, S, B, T\rangle_n \rightarrowtail \langle G, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$ where c is a CHR constraint.
3. Apply $\langle G, H_1 \uplus H_2 \uplus S, B, T\rangle_n \rightarrowtail \langle C \uplus G, H_1 \uplus S, \theta \wedge B, T'\rangle_n$ where there exists a (renamed apart) rule in P of the form

$$r \ @ \ H'_1 \setminus H'_2 \iff g \mid C$$

and a matching substitution θ such that $chr(H_1) = \theta(H'_1)$, $chr(H_2) = \theta(H'_2)$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+ \ id(H_2) +\!\!+ \ [r] \notin T$. In the result $T' = T \cup \{id(H_1) +\!\!+ \ id(H_2) +\!\!+ \ [r]\}$.[5] $\square$

The first rule tells the underlying solver to add a new builtin constraint to the builtin constraint store. The second adds a new identified CHR constraint to the CHR constraint store. The last one chooses a program rule for which matching constraints exist in the CHR constraint store, and whose guard is entailed by the underlying solver, and fires it. For readability, we usually apply the resulting substitution θ to all relevant fields in the execution state, i.e. G, S and B. This does not affect the meaning of the execution state, or its transition applicability, but it helps remove the build-up of too many variables and constraints.

The transitions are non-deterministically applied until either no more transitions are applicable (a successful derivation), or the underlying solver can prove $\mathcal{D} \models \neg\bar{\exists}_\emptyset B$ (a failed derivation). In both cases a *final state* has been reached.

Example 5. The following is a (terminating) derivation under ω_t for the query `gcd(6), gcd(9)` executed on the `gcd` program in Example 4. For brevity, B and

[5] Note in practice we only need to keep track of tuples where H_2 is empty, since otherwise these CHR constraints are being deleted and the firing can not reoccur.

T have been removed from each tuple.

$$
\begin{array}{lll}
 & & \langle\{\texttt{gcd}(6), \texttt{gcd}(9)\}, \emptyset\rangle_1 \quad (1)\\
 & \rightarrowtail_{introduce} & \langle\{\texttt{gcd}(9)\}, \{\texttt{gcd}(6)\#1\}\rangle_2 \quad (2)\\
 & \rightarrowtail_{introduce} & \langle\emptyset, \{\texttt{gcd}(6)\#1, \texttt{gcd}(9)\#2\}\rangle_3 \quad (3)\\
(\texttt{gcd2}\ \ N = 6 \wedge M = 9) & \rightarrowtail_{apply} & \langle\{\texttt{gcd}(3)\}, \{\texttt{gcd}(6)\#1\}\rangle_3 \quad (4)\\
 & \rightarrowtail_{introduce} & \langle\emptyset, \{\texttt{gcd}(6)\#1, \texttt{gcd}(3)\#3\}\rangle_4 \quad (5)\\
(\texttt{gcd2}\ \ N = 3 \wedge M = 6) & \rightarrowtail_{apply} & \langle\{\texttt{gcd}(3)\}, \{\texttt{gcd}(3)\#3\}\rangle_4 \quad (6)\\
 & \rightarrowtail_{introduce} & \langle\emptyset, \{\texttt{gcd}(3)\#3, \texttt{gcd}(3)\#4\}\rangle_5 \quad (7)\\
(\texttt{gcd2}\ \ N = 3 \wedge M = 3) & \rightarrowtail_{apply} & \langle\{\texttt{gcd}(0)\}, \{\texttt{gcd}(3)\#3\}\rangle_5 \quad (8)\\
 & \rightarrowtail_{introduce} & \langle\emptyset, \{\texttt{gcd}(3)\#3, \texttt{gcd}(0)\#5\}\rangle_6 \quad (9)\\
(\texttt{gcd1}) & \rightarrowtail_{apply} & \langle\emptyset, \{\texttt{gcd}(3)\#3\}\rangle_6 \quad (10)
\end{array}
$$

No more transition rules are possible, so this is the final state.

3 The Refined Operational Semantics ω_r

The *refined* operational semantics establishes an order for the constraints in G. As a result, we are no longer free to pick any constraint from G to either **Solve** or **Introduce** into the store. It also treats CHR constraints as procedure calls: each newly added *active* constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires other CHR constraints might be executed and, when they finish, the execution returns to finding rules for the current active constraint. Not surprisingly, this approach is used exactly because it corresponds closely to that of the language we compile to.

Formally, the execution state of the refined semantics is the tuple $\langle A, S, B, T\rangle_n$ where S, B, T and n, representing the CHR store, builtin store, propagation history and next free identity number respectively, are exactly as before. The *execution stack* A is a sequence of constraints, identified CHR constraints and occurrenced identified CHR constraints, with a strict ordering in which only the top-most constraint is active. An *occurrenced* identified CHR constraint $c\#i : j$ indicates that only matches with occurrence j of constraint c should be considered when the constraint is active. Unlike in the theoretical operational semantics, the same identified constraint may simultaneously appear in both the execution stack A and the store S.

Given initial goal G, the initial state is as before. Just as with the theoretical operational semantics, execution proceeds by exhaustively applying transitions to the initial execution state until the builtin solver state is unsatisfiable or no transitions are applicable. The possible transitions are as follows:

1. Solve $\langle [c|A], S_0 \uplus S_1, B, T\rangle_n \rightarrowtail \langle S_1 +\!\!+ A, S_0 \uplus S_1, c \wedge B, T\rangle_n$ where c is a builtin constraint, and $vars(S_0) \subseteq fixed(B)$, where $fixed(B)$ is the set of variables fixed by B.[6] This reconsiders constraints whose matches might be affected by c.
2. Activate $\langle [c|A], S, B, T\rangle_n \rightarrowtail \langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$ where c is a CHR constraint (which has never been active).

[6] $v \in fixed(B)$ if $\mathcal{D} \models \bar{\exists}_v(B) \wedge \bar{\exists}_{\rho(v)}\rho(B) \rightarrow v = \rho(v)$ for arbitrary renaming ρ.

3. Reactivate $\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : 1|A], S, B, T\rangle_n$ where c is a CHR constraint (re-added to A by **Solve** but not yet active).
4. Drop $\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle A, S, B, T\rangle_n$ where $c\#i : j$ is an occurrenced active constraint and there is no such occurrence j in P (all existing ones have already been tried thanks to transition 7).
5. Simplify $\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail \langle C +\!\!+ A, H_1 \uplus S, \theta \wedge B, T'\rangle_n$ where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \;@\; H'_1 \setminus H'_2, d_j, H'_3 \iff g \mid C$$

and there exists matching substitution θ is such that $c = \theta(d_j)$, $chr(H_1) = \theta(H'_1)$, $chr(H_2) = \theta(H'_2)$, $chr(H_3) = \theta(H'_3)$, and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T$. In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$.
6. Propagate $\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail \langle C +\!\!+ [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, \theta \wedge B, T'\rangle_n$ where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \;@\; H'_1, d_j, H'_2 \setminus H'_3 \iff g \mid C$$

and there exists matching substitution θ is such that $c = \theta(d_j)$, $chr(H_1) = \theta(H'_1)$, $chr(H_2) = \theta(H'_2)$, $chr(H_3) = \theta(H'_3)$,and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T$. In the result $T' = T \cup \{id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r]\}$.

The role of the propagation histories T and T' is exactly the same as with the theoretical operational semantics, ω_t.
7. Default $\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : j + 1|A], S, B, T\rangle_n$ if the current state cannot fire any other transition. $\square$

The refined operational semantics is still *non-deterministic*. Its first source of non-determinism is the **Solve** transition where the order in which constraints S_1 are added to the activation stack is still left open. The definition above (which considers all non-fixed CHR constraints) is weak. In practice, only constraints that may potentially cause a new rule to fire are re-added, see [5, 10] for more details.

The other source of non-determinism occurs within the **Simplify** and **Propagate** transitions, where we do not know which *partner* constraints (H_1, H_2 and H_3) may be chosen for the transition, if more than one possibility exists.

Both sources of non-determinism could be removed by further refining the operational semantics, however we use non-determinism to model implementation specific behaviour of CHRs. For example, different CHR implementations use different data structures to represent the store, and this may inadvertently affect the order partner constraints are matched against a rule. By leaving matching order non-deterministic, we capture the semantics of all current implementations. It also leave more freedom for optimization of CHR execution (see e.g. [12]).

Example 6. The following shows the derivation under ω_r semantics for the `gcd` program in Example 4 and the goal `gcd(6),gcd(9)`. For brevity B and T have been eliminated and the substitutions θ applied throughout.

$$
\begin{array}{lll}
 & \langle [\mathtt{gcd}(6), \mathtt{gcd}(9)], \emptyset\rangle_1 & (1) \\
\rightarrowtail_{activate} & \langle [\mathtt{gcd}(6)\#1:1, \mathtt{gcd}(9)], \{\mathtt{gcd}(6)\#1\}\rangle_2 & (2) \\
\rightarrowtail^{\times 3}_{default} & \langle [\mathtt{gcd}(6)\#1:4, \mathtt{gcd}(9)], \{\mathtt{gcd}(6)\#1\}\rangle_2 & (2) \\
\rightarrowtail_{drop} & \langle [\mathtt{gcd}(9)], \{\mathtt{gcd}(6)\#1\}\rangle_2 & (2) \\
\rightarrowtail_{activate}\rightarrowtail_{default} & \langle [\mathtt{gcd}(9)\#2:2], \{\mathtt{gcd}(9)\#2, \mathtt{gcd}(6)\#1\}\rangle_3 & (3) \\
\rightarrowtail_{simplify} & \langle [\mathtt{gcd}(3)], \{\mathtt{gcd}(6)\#1\}\rangle_3 & (4) \\
\rightarrowtail_{activate}\rightarrowtail^{\times 2}_{default} & \langle [\mathtt{gcd}(3)\#3:3], \{\mathtt{gcd}(3)\#3, \mathtt{gcd}(6)\#1\}\rangle_3 & (5) \\
\rightarrowtail_{propagate} & \langle [\mathtt{gcd}(3), \mathtt{gcd}(3)\#3:3], \{\mathtt{gcd}(3)\#3\}\rangle_4 & (6) \\
\rightarrowtail_{activate}\rightarrowtail_{default} & \langle [\mathtt{gcd}(3)\#4:2, \mathtt{gcd}(3)\#3:3], \{\mathtt{gcd}(3)\#4, \mathtt{gcd}(3)\#3\}\rangle_5 & (7) \\
\rightarrowtail_{simplify} & \langle [\mathtt{gcd}(0), \mathtt{gcd}(3)\#3:3], \{\mathtt{gcd}(3)\#3\}\rangle_5 & (8) \\
\rightarrowtail_{activate} & \langle [\mathtt{gcd}(0)\#5:1, \mathtt{gcd}(3)\#3:3], \{\mathtt{gcd}(0)\#5, \mathtt{gcd}(3)\#3\}\rangle_6 & (9) \\
\rightarrowtail_{simplify} & \langle [\mathtt{gcd}(3)\#3:3], \{\mathtt{gcd}(3)\#3\}\rangle_6 & (10) \\
\rightarrowtail_{default}\rightarrowtail_{drop} & \langle \epsilon, \{\mathtt{gcd}(3)\#3\}\rangle_6 & (10)
\end{array}
$$

4 The Relationship between the Two Semantics

Once both semantics are established, we can define an abstraction function α which maps execution states of ω_r to ω_t as follows:

$$\alpha(\langle A, S, B, T\rangle_n) = \langle no\_id(A), S, B, T\rangle_n$$

where $no\_id(A) = \{c \mid c \in A \text{ is not of the form } c\#i \text{ or } c\#i:j\}$.

Example 7. A state in Example 6 with number (N) is mapped to the state in Example 5 with the same number. For example, the state $\langle [\mathtt{gcd}(0), \mathtt{gcd}(3)\#3 : 3], \{\mathtt{gcd}(3)\#3\}\rangle_5$ corresponds to $\langle \{\mathtt{gcd}(0)\}, \{\mathtt{gcd}(3)\#3\}\rangle_5$ since both are numbered (8).

We now extend α to map a derivation D in ω_r to the corresponding derivation $\alpha(D)$ in ω_t, by mapping each state appropriately and eliminating adjacent equivalent states:

$$\alpha(S_1 \rightarrowtail D) = \begin{array}{ll} \alpha(D) & \text{if } D = S_2 \rightarrowtail D' \text{ and } \alpha(S_1) = \alpha(S_2) \\ \alpha(S_1) \rightarrowtail \alpha(D) & \text{otherwise} \end{array}$$

We can now show that each ω_r derivation has a corresponding ω_t derivation, and the final state of the ω_r corresponds to a final state in the ω_t derivation.

Theorem 1 (Correspondence). *Given a derivation D under ω_r then there exists a corresponding derivation $\alpha(D)$ under ω_t. If S is the final state in D then $\alpha(S)$ is a final state under ω_t.*

Theorem 1 shows that the refined operational semantics implements the theoretical operational semantics. Hence, the soundness and completeness results for CHRs under the theoretical operational semantics hold under the refined operational semantics ω_r.

4.1 Termination

Termination of CHR programs is obviously a desirable property. Thanks to Theorem 1, termination of ω_t programs ensures termination of ω_r.

Corollary 1. *If every derivation for G in ω_t terminates, then every derivation for G in ω_r also terminates.*

The converse is clearly not true, as shown in Example 1. In practice, proving termination for CHR programs under the theoretical operational semantics is quite difficult (see [8] for examples and discussion). It is somewhat simpler for the refined operational semantics but, just as with other programming languages, this is simply left to the programmer.

4.2 Confluence

Since both operational semantics of CHRs are non-deterministic, *confluence* of the program, which guarantees that whatever order the rules are applied in leads to the same result, is essential from a programmer's point of view. Without it the programmer cannot anticipate the answer that will arise from a goal.

Formally, a CHR program P is confluent under semantics ω if for any goal G and any two derivations $\langle G, \emptyset, true, \emptyset\rangle_1 \rightarrowtail^*_\omega \langle \_, S_1, B_1, \_\rangle_\_$ and $\langle G, \emptyset, true, \emptyset\rangle_1 \rightarrowtail^*_\omega \langle \_, S_2, B_2, \_\rangle_\_$ we have that $\mathcal{D} \models \bar{\exists}_G(S_1 \wedge B_1) \leftrightarrow \bar{\exists}_G(S_2 \wedge B_2)$. That is, the resulting constraints stores are equivalent.

Confluence of the theoretical operational semantics of CHR programs has been extensively studied [1, 2]. Abdennadher [1] provides a decidable confluence test for the theoretical semantics of terminating CHR programs. Essentially, it relies on computing critical pairs where two rules can possibly be used, and showing that each of the two resulting states lead to equivalent states.

Just as with termination, thanks to Theorem 1, confluence under ω_t implies confluence under ω_r.

Corollary 2. *If CHR program P is confluent with respect to ω_t, it is confluent with respect to ω_r.*

5 Checking Confluence for ω_r

One of the benefits of exposing the refined operational semantics is the ability to write and execute programs that are non-confluent with respect to the theoretical operational semantics, but are confluent with respect to the refined operational semantics. In order to take advantage of this, we need to provide a decidable test for confluence under ω_r. This test must be able to capture a reasonable number of programs which are confluent under ω_r but not under ω_t. However, this appears to be quite difficult.

Example 8. For example, consider the following CHR program

```
p1 @ p <=> true.
p2 @ q(_), p <=> true.
```

Rule `p2` looks suspiciously non-confluent since, if it was the only rule present, the goal `q(a),q(b),p` could terminate with either `q(a)` or `q(b)` left in the store. However, when combined with `p1`, `p2` will never fire since any active `p` constraint will be deleted by `p1` before reaching `p2`. Thus, the program is ω_r confluent.

The example illustrates how extending the notion of critical pairs can be difficult, since many critical pairs will correspond to unreachable program states.

As mentioned before, there are two sources of non-determinism in the refined operational semantics. The first source, which occurs when deciding the order in which the CHR constraints are added to the activation stack while applying **Solve**, is hard to tackle. In practice, we will avoid re-activating most CHR constraints in the store, by only considering those which might now cause a rule to fire when it did not fire before (see [5, 10] for more details). However, if re-activation actually occurs, the programmer is unlikely to have any control on what order re-activated constraints are re-executed. To avoid this non-determinism we will require S_1 to be empty in any **Solve** transition. This has in fact been the case for all the examples considered so far except `fib`, and all those in Section 6, since all CHR constraints added to the store had fixed arguments. Even for `fib` we could safely avoid reactivating the `fib` constraints whose second arguments are not fixed, since these arguments have no relationship with the guards.

For programs that really do interact with an underlying constraint solver, we have no better solution than relying on the confluence test of the theoretical operational semantics, for in this case it is very hard to see how the programmer can control execution sufficiently.

The second source of non-determinism occurs when there is more than one set of partner constraints in the CHR store that can be used to apply the **Simplify** and **Propagate** transitions. We formalise this as follows. A *matching* of occurrence j with active CHR constraint c in state $\langle [c\#i : j|A], S, B, T\rangle_n$ is the sequence of identified constraints $H_1 +\!\!+ H_2 +\!\!+ H_3 +\!\!+ [c\#i]$ used in transitions **Simplify** and **Propagate**. The *goal* of the matching is the right hand side of the associated rule with the matching substitution applied, i.e., $\theta(C)$.

Non-confluence arises when multiple matchings exist for a rule R, and R is not allowed to eventually try them all. This can happen if firing R with one matching results in the deletion of a constraint in another matching.

Definition 1. *An occurrence j in rule R is* matching complete *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, ..., M_m$ possible matchings and $G_1, \ldots, G_m$ corresponding goals, firing R for any matching M_l and executing G_l does not result in the deletion of a constraint occurring in a different matching $M_k, k \neq l$.*

Note that R itself may directly delete the active constraint. If so, R will only be matching complete if there is only one possible matching, i.e., $m = 1$.

Example 9. Consider the CHR program in Example 2. The occurrence of `entry` in rule `l1` is matching complete since the `lookup` constraint is never stored (it is deleted before it becomes inactive). This is not however the case for the occurrence of `lookup` in `l1`. Goal `entry(a,b),entry(a,c),lookup(a,V)` will return `V=b` or `V=c` depending on which of the two matchings of the occurrence of `lookup` in `l1` (`[entry(a,b)#1,lookup(a,V)#3]` or `[entry(a,c)#2,lookup(a,V)#3]`) is used, i.e., depending on which partner `entry` constraint is used for `lookup(a,V)` in `l1`. The code is matching complete if the database only contains one entry per key. Adding the rule

```
killdup @ entry(Key,Val1) \ entry(Key,Val2) <=> true.
```

which throws away duplicate entries for a key, provides a functional dependency from `Key` to `Val` in `entry(Key,Val)`. This rule makes the occurrence matching complete, since only one matching will ever be possible.

Matching completeness can also be broken if the body of a rule deletes constraints from other matchings.

Example 10. Consider the following CHR program

```
r1 @ p, q(X) ==> r(X).
r2 @ p, r(a) <=> true.
```

The occurrence of `p` in `r1` is not matching complete since the goal `q(a),q(b),p`, will obtain the final state `q(a),q(b)` or `q(a),q(b),r(b)` depending on which partner constraint (`q(a)` or `q(b)`) is used for the occurrence of `p` in `r1`. This is because the goal of the first matching (`r(a)`) deletes `p`.

A matching complete occurrence is guaranteed to eventually try all possible matchings for given execution state S. However, matching completeness is sometimes too strong if the user doesn't care which matching is chosen. This is common when the body does not depend on the matching chosen.

Example 11. For example, consider the following rule from a simple ray tracer.

```
shadow @ sphere(C,R,_) \ light_ray(L,P,_,_) <=> blocks(L,P,C,R) | true.
```

This rule calculates if point `P` is in shadow by testing if the ray from light `L` is blocked by a sphere at `C` with radius `R`. Consider an active `light_ray` constraint, there may be more than one `sphere` blocking the ray, however we don't care *which* sphere blocks, just *if* there is a sphere which blocks. This rule is not matching complete but, since the matching chosen does not affect the resulting state, it is matching independent.

Definition 2. *A matching incomplete occurrence j which is deleted by rule R is* matching independent *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, \ldots, M_m$ possible matchings and $G_1, \ldots, G_m$ corresponding goals, then all the final states for $\langle G_k, S_k, B, T_k\rangle_n, 1 \leq k \leq m$ are equivalent, where S_k is the store after firing on matching M_k and T_k is the resulting history.*

Suppose that a rule is matching complete, and there are multiple possible matchings. The ordering in which the matchings are tried is still chosen non-deterministically. Hence, there is still potential of non-confluence. For this reason we also require *order independence*, which ensures the choice of order does not affect the result.

Definition 3. *A matching complete occurrence j in rule R is* order independent *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, \ldots, M_m$ possible matchings and $G_1, \ldots, G_m$ corresponding goals, the execution of the state $\langle G_{\sigma(1)} +\!\!+ \cdots +\!\!+ G_{\sigma(m)}, S', B, T'\rangle_n$ where S' is the CHR store S where all constraints deleted by any matching are deleted, and T' has all sequences added by all matchings, for any permutation σ, leads to equivalent states.*

Note that, since j is matching complete, S' is well defined. Order independence is a fairly strong condition and, currently, we have little insight as to how to check it beyond a limited version of the confluence check for the theoretical operational semantics. Thus, we currently require user annotations about order independence. A matching complete occurrence, which may have more than one matching, only passes the confluence checker if all CHR constraints called by the body are annotated as order independent.

Example 12. Consider the following fragment for summing colors from a ray tracer.

```
add1 @ add_color(C1), color(C2) <=> C3 = C1 + C2, color(C3).
add2 @ add_color(C) <=> color(C).
```

All occurrences of `color` and `add_color` are matching complete. Furthermore, calling `add_color(`C_1`), ..., add_color(`C_n`)` results in `color(`C_1+...+C_n`)`. Since addition is symmetric and associative, it does not matter in what order the `add_color` constraints are called. Consider the occurrence of `output` in

```
render @ output(P) \ light_ray(_,P,C,_) <=> add_color(C).
```

Here, calling `output(P)` calculates the (accumulated) color at point `P` where any `light_ray`s (a ray from a light source) may intersect. If there are multiple light sources, then there may be multiple `light_ray` constraints. The order `add_color` is called does not matter, hence the occurrence is order independent.

We now have sufficient conditions for a simple confluence test.

Theorem 2 (Confluence Test). *Let P be a CHR program such that:*

1. *Starting from a fixed goal, any derived state is also fixed;*
2. *All occurrences in rules are matching complete or matching independent;*
3. *All matching complete occurrences in rules are order independent.*

Then P is ω_r confluent for fixed goals.

The HAL CHR confluence checker implements partial tests for fixedness of CHR constraints, matching completeness and matching independence, and relies on user annotation for determining order independence.

The confluence checker uses mode checking [9] to determine which CHR constraints are always fixed. A non-fixed constraint may also be safe, as long as it is never in the store when it is not active (such as `lookup` from Example 2). We call such constraints *never stored*.

The confluence checker uses information about never stored constraints and functional dependency analysis (see [12]) to determine how many possible matchings (0, 1 or many) there are for each occurence in a given rule. If there are multiple possible matchings for an occurence, it then checks that the removal of other matching constraints is impossible, by examining the rule itself and using a reachability analysis of the "call graph" for CHR rules, to determine if the constraints could be removed by executing the body of the rule.

The checker determines matching independence by determining which variables occuring in the body are functionally defined by the active occurence, making use of functional dependency analysis to do so. If all variables in the body and the deleted constraints are functionally defined by the active occurence, the occurence is matching independent.

Only bodies restricted to built-in constraints are considered as order independent by the current confluence checker. Otherwise, we rely on user annotation.

6 Case Studies: Confluence Test

This section investigates the confluence of two "real-life" CHR programs using our confluence checker. The programs are `bounds` – an extensible bounds propagation solver, and `compiler` – a new (bootstrapping) CHR compiler. Both were implemented with the refined operational semantics in mind, and simply will not work under the theoretical semantics.

6.1 Confluence of `bounds`

The bounds propagation solver is implemented in HAL and has a total of 83 rules, 37 CHR constraints and 113 occurences. An early version of a bounds propagation solver first appeared in [12]. The current version also implements simple dynamic scheduling (i.e. the user can delay goals until certain conditions hold), as well as supporting ask constraints. This program was implemented before the confluence checker.

The confluence checker finds 4 matching problems, and 3 order independence problems. One of the matching problems indicated a bug (see below), the others are attributed to the weakness in the compiler's analysis. We only had to annotate one constraint as order independent.

The confluence analysis complained that the following rule is matching incomplete and non-independent when `kill(Id)` is active since there are (potentially) many possible matchings for the `delayed_goals` partner.

```
kill @ kill(Id), delayed_goals(Id,X,_,...,_) <=> true.
```

Here `delayed_goals(Id,X,_,...,_)` represents the delayed goals for bounds solver variable X. The code should be

```
kill1 @ kill(Id) \ delayed_goals(Id,X,_,...,_) <=> true.
kill2 @ kill(_) <=> true.
```

This highlights how a simple confluence analysis can be used to discover bugs.

The confluence analysis also complains about the rules for bounds propagation themselves. The reason is that the constraint `bounds(X,L,U)` which stores the lower L and upper U bounds of variable X has complex self-interaction. Two `bounds` constraints for the same variable can interact using, for example,

```
b2b @ bounds(X,L1,U1), bounds(X,L2,U2) <=> bounds(X,max(L1,L2),min(U1,U2)).
```

Here, the user must annotate the matching completeness and order independence of `bounds`. In fact, the relevant parts of the program are confluent within the theoretical operational semantics, but this is currently beyond the capabilities of our confluence analysis (and difficult because it requires bounds reasoning).

6.2 Confluence of `compiler`

The bootstrapping compiler is implemented in SICStus Prolog (using the CHR library), including a total of 131 rules, 42 CHR constraints and 232 occurences. and performs most of the analysis and optimisations detailed in [12]. After bootstrapping it has similar speed to the original compiler written in Prolog and produces more efficient code due to the additional analysis performed. During the bootstrap, when compiling itself the first time, the new code outperformed the old code (the SICStus Prolog CHR compiler, 1100 lines of Prolog) by a factor of five. This comparison is rather crude, measuring the costs and effects of the optimisations based on the additional analysis and the improved runtime system at once. Yet it demonstrates the practicality of the bootstrapping approach for CHRs and that CHRs as a general purpose programming language under the refined semantics can be used to write moderately large sized verifiable programs.

Bootstrapping CHRs as such aims at easier portability to further host languages and as an internal reality check for CHRs as a general purpose programming system. To the best of our knowledge, the bootstrapping compiler is the largest single CHR program written by hand. (Automatic rule generators for constraint propagation algorithms [3] can produce large CHR programs too, but from the point of the compiler their structure is rather homogeneous in comparison to the compiler's own code).

The confluence checker finds 14 matching problems, and 45 order independence problems. 4 of the matching problems are removed by making functional dependencies explicit. The others are attributed to the weakness in the compiler's analysis. We had to annotate 18 constraints as order independent.

7 Conclusion

The refined operational semantics for Constraint Handling Rules provides a powerful and expressive language, ideal for applications such as compilers, since fixpoint computations and simple database operations are straightforward to program. In order to support programming for this language we need to help the author check the confluence of his program. In this paper we have defined a partial confluence checker that is powerful enough to check many idioms used in real programs. In the future we intend to extend this checker to better handle order independence, and to include the ability to check confluence with respect to the theoretical semantics.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 252–266, 1997.
2. S. Abdennadher, T. Frühwirth, and H. Muess. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–166, 1999.
3. K. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Principles and Practice of Constraint Programming*, pages 58–72, 1999.
4. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, pages 174–188, 1999.
5. G.J. Duck, P.J. Stuckey, M. García de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In D. Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 79–90. ACM Press, 2003.
6. T. Frühwirth. `http://www.pms.informatik.uni-muenchen.de/~webchr/`.
7. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37:95–138, 1998.
8. T. Frühwirth. Proving termination of constraint solver programs. In *New Trends in Contraints, Joint ERCIM/Compulog Net Workshop*, number 1865 in LNCS, pages 298–317. Springer-Verlag, 1999.
9. M. García de la Banda, P.J. Stuckey, W. Harvey, and K. Marriott. Mode checking in HAL. In J. LLoyd et al., editor, *Proceedings of the First International Conference on Computational Logic*, LNCS 1861, pages 1270–1284. Springer-Verlag, July 2000.
10. C. Holzbaur and T. Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 117–133. Springer-Verlag, 1999.
11. C. Holzbaur and T. Frühwirth. A Prolog constraint handling rules compiler and runtime system. *Journal of Applied Artificial Intelligence*, 14(4), 2000.
12. C. Holzbaur, P.J. Stuckey, M. García de la Banda, and D. Jeffery. Optimizing compilation of constraint handling rules. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS, pages 74–89. Springer-Verlag, 2001.