A State Equivalence and Confluence Checker for CHR

Johannes Langbein, Frank Raiser, and Thom Frühwirth

Faculty of Engineering and Computer Science, Ulm University, Germany
`firstname.lastname@uni-ulm.de`

Abstract. Analyzing confluence of CHR programs manually can be an impractical and time consuming task. Based on a new theorem for state equivalence, this work presents the first tool for testing equivalence of CHR states. As state equivalence is an essential component of confluence analysis, we apply this tool in the development of a confluence checker that overcomes limitations of existing checkers. We further provide evaluation results for both tools and detail their modular design, which allows for extensions and reuse in future implementations of CHR tools.

1 Introduction

Constraint Handling Rules (CHR) [1] is a declarative, multiset- and rule-based programming language suitable for powerful program analysis.

One such property a CHR program can be analyzed for is called *confluence*. For a confluent program, it is guaranteed that we always get the same result for a given query, independently of the order of rule applications and the order of constraints in the query. Furthermore, confluent programs are parallelizable without changing their source code [1].

There is a decidable, sufficient, and necessary criterion for confluence of terminating CHR programs [2]. It is based on the joinability of so-called *critical pairs*. However, proving confluence of a larger program manually can quickly become infeasible as there is a combinatorial explosion in the number of critical pairs with program size. Hence, confluence checking is preferably done using a software tool.

Equivalence of CHR states is a fundamental notion used in the criterion for confluence. Recent work [3,4] has led to an axiomatic definition of state equivalence alongside a decidable and sufficient criterion.

Based on this work, we implemented a checker for equivalence of CHR states. Furthermore, we used this checker to implement a new tool for confluence checking, which overcomes limitations (cf. Section 4.2) of existing confluence checkers [2,5]. The checkers for state equivalence and confluence together with some example CHR programs and test-cases are available under GNU GPL and can be downloaded from [6]. They can be used for any terminating CHR program which does not contain propagation rules. Furthermore, only the built-ins `true`, `false`, and `=/2` are supported (cf. Section 5.1).

We make the following contributions:

- We present the first implementation of a test for CHR state equivalence (Section 3.1).
- Based on this, we describe our implementation of a new confluence checker (Section 3.2).
- We evaluate our confluence checker with the union-find implementations presented in [5] and compare it to previous implementations (Section 4) .
- Finally, we point out possible future extensions of our checkers (Section 5.1).

2 Preliminaries

Constraint Handling Rules distinguishes two kinds of constraints: *CHR constraints* and *built-in constraints*. We assume reasoning on built-in constraints to be possible through a satisfaction-complete and decidable constraint theory $\mathcal{CT}$. We use the following definitions of CHR states and state equivalence from [3].

Definition 1 (CHR States). *A CHR state σ is a tuple $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$. The* goal $\mathbb{G}$ *is a multiset of CHR constraints. The* built-in constraint store $\mathbb{B}$ *is a conjunction of built-in constraints. $\mathbb{V}$ is a set of* global variables. *A variable $v \in (\mathbb{B} \cup \mathbb{G})$ is called a* local variable *iff $v \notin \mathbb{V}$. A variable $v \in \mathbb{B}$ is called a* strictly local variable *iff $v \notin (\mathbb{V} \cup \mathbb{G})$. We use Σ to denote the set of all states.*

Definition 2 (State Equivalence). *Equivalence between CHR states is the smallest equivalence relation $\equiv$ over CHR states that satisfies the following conditions:*

1. (Equality as Substitution)
$\langle \mathbb{G}; X \doteq t \wedge \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G}[X/t]; X \doteq t \wedge \mathbb{B}; \mathbb{V} \rangle$
2. (Transformation of the Constraint store)
If $\mathcal{CT} \models \forall(\exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}')$ where $\bar{s}, \bar{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$, respectively, then $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}'; \mathbb{V} \rangle$
3. (Omission of Non-Occuring Global Variables)
If X is a variable that does not occur in $\mathbb{G}$ or $\mathbb{B}$ then $\langle \mathbb{G}; \mathbb{B}; \{X\} \cup \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$
4. (Equivalence of failed states)
$\langle \mathbb{G}; \bot; \mathbb{V} \rangle \equiv \langle \mathbb{G}'; \bot; \mathbb{V} \rangle$

The following sufficient and decidable criterion for state equivalence was introduced in [3].

Theorem 1 (Criterion for $\equiv$). *Let $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ and $\sigma' = \langle \mathbb{G}'; \mathbb{B}'; \mathbb{V} \rangle$ be CHR states with local variables $\bar{y}$ and $\bar{y}'$ that have been renamed apart. Then*

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

For $\mathbb{G} = \{c_1, \ldots, c_n\}$ and $\mathbb{G}' = \{c'_1, \ldots, c'_n\}$ the expression $(\mathbb{G} = \mathbb{G}')$ denotes the following set of equations: $c_{\tau(1)} = c'_1 \wedge \ldots \wedge c_{\tau(n)} = c'_n$ for a permutation τ.

A CHR program $\mathcal{P}$ is a set of rules of the following form.

Definition 3 (CHR Rule). *A CHR (simpagation) rule is of the form*

$$r \ @ \ H_1 \backslash H_2 \Leftrightarrow G \mid B_c \uplus B_b$$

where H_1 and H_2 are multisets of user-defined constraints, called the kept head *and* removed head*, respectively. The* guard *G is a conjunction of built-in constraints and the* body *consists of a conjunction of built-in constraints B_b and a multiset of user-defined constraints B_c. The* rule name *r is optional and may be omitted along with the @ symbol. If H_1 is empty, we call the rule a* simplification *rule.*

In this paper, we use the following *equivalence-based operational semantics ω_e* which was established in [3].

Definition 4 (ω_e Transitions). *For a CHR program $\mathcal{P}$, the state transition system $(\Sigma/\equiv, \rightarrowtail)$ is defined as follows. The transition is based on a variant of a rule r in $\mathcal{P}$ such that its local variables are disjoint from the variables occurring in the pre-transition state.*

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

When the rule r is clear from the context or not important, we may write $\rightarrowtail$ rather than $\rightarrowtail^r$. By $\rightarrowtail^*$, we denote the reflexive-transitive closure of $\rightarrowtail$. For two states σ and σ' we may also write $\sigma \rightarrowtail \sigma'$ instead of $[\sigma] \rightarrowtail [\sigma']$.

Confluence is a property which guarantees that all possible computations for a given goal result in equivalent final states. For confluent programs, the order of constraints in a goal and the order of rules in the program does not matter [2]. We define confluence using the following definition of joinability.

Definition 5 (Joinability). *Two CHR states σ_1 and σ_2 are called* joinable *if there exist states σ_1' and σ_2' such that $\sigma_1 \rightarrowtail^* \sigma_1'$ and $\sigma_2 \rightarrowtail^* \sigma_2'$ with $\sigma_1' \equiv \sigma_2'$*

Definition 6 (Confluence). *A CHR program is* confluent *if for all states σ, σ_1, σ_2:*

$$\text{If } \sigma \rightarrowtail^* \sigma_1 \text{ and } \sigma \rightarrowtail^* \sigma_2, \text{ then } \sigma_1 \text{ and } \sigma_2 \text{ are joinable.}$$

We cannot check joinability starting from all possible states σ, as in general there are infinitely many such states. However, there exists a decidable, sufficient and necessary criterion for confluence of terminating CHR programs [2], which is based on joinability of so-called critical pairs. The criterion states, that for terminating programs, we can restrict the joinability test to those critical pairs, of which only a finite number exists. For critical pairs, we use the following, adapted definition from [7]:

Definition 7 (Critical Pair). *Given two (not necessarily different) rules $r1$ @ $H_{11} \backslash H_{21} \Leftrightarrow G_1 \mid B_{c1} \uplus B_{b1}$ and $r2$ @ $H_{12} \backslash H_{22} \Leftrightarrow G_2 \mid B_{c2} \uplus B_{b2}$ whose variables have been renamed apart.*
The tuple (σ_1, σ_2) with

$$\sigma_1 = \langle B_{c1} \uplus ((H_{r1}^{\triangle} \uplus H_{r2}^{\triangle} \uplus H_{r1}^{\cap}) - H_{21}); (H_{r1}^{\cap} = H_{r2}^{\cap}) \wedge B_{b1} \wedge G_1 \wedge G_2; \mathbb{V}\rangle$$

$$\sigma_2 = \langle B_{c2} \uplus ((H_{r1}^{\triangle} \uplus H_{r2}^{\triangle} \uplus H_{r1}^{\cap}) - H_{22}); (H_{r1}^{\cap} = H_{r2}^{\cap}) \wedge B_{b2} \wedge G_1 \wedge G_2; \mathbb{V}\rangle$$

where

$$H_{r1}^{\triangle} \uplus H_{r1}^{\cap} = H_{11} \uplus H_{21}$$

$$H_{r2}^{\triangle} \uplus H_{r2}^{\cap} = H_{12} \uplus H_{22}$$

$$\mathbb{V} = vars(H_{r1}^{\triangle} \wedge H_{r2}^{\triangle} \wedge H_{r1}^{\cap} \wedge G_1 \wedge G_2)$$

is called a critical pair *of r_1 and r_2 if $H_{r1}^{\cap} \neq \emptyset$ and $\mathcal{CT} \models (H_{r1}^{\cap} = H_{r2}^{\cap}) \wedge G_1 \wedge G_2$.*

Here, the equality $=$ between multisets of head constraints has the same meaning as in Theorem 1.

It suffices to show joinability of critical pairs to show confluence of a terminating program.

Theorem 2 (Criterion for confluence [2]). *A terminating CHR program $\mathcal{P}$ is confluent iff all critical pairs of all rules of $\mathcal{P}$ are joinable.*

In [2] this theorem has been proven for CHR programs consisting only of simplification rules. However, as we do not consider propagation rules and as simpagation rules are only an abbreviation for simplification rules, the theorem remains valid. Furthermore, this allows us to ignore the token-store from [8] and thus apply the theorem using CHR states according to Definition 1.

Confluence can be tested by posing the two states σ_1 and σ_2 of a critical pair as query to the CHR program, as the following argumentation shows: If the two queries result in two states σ_1' and σ_2' with $\sigma_1' \equiv \sigma_2'$, we have shown joinability of the critical pair. If the two queries result in two non-equivalent states σ_1' and σ_2', there should be transitions $\sigma_1 \rightarrowtail^* \sigma_1''$ and $\sigma_2' \rightarrowtail^* \sigma_2''$ such that $\sigma_1'' \equiv \sigma_2''$, if the program was confluent as the order of rule application should not matter in this case. However, if the computation stops with σ_1' and σ_2' as final states, no more rules are applicable, in particular there are no transitions leading to equivalent states σ_1'' and σ_2''. This means we have found a counterexample for confluence of the program.

3 State Equivalence and Confluence Checkers

The checkers for state equivalence and confluence are implemented in SWI-Prolog. They are organized in two different modules named `stateequiv` and `conflcheck` which define the predicates for state equivalence and confluence, respectively.

3.1 State Equivalence

The module `stateequiv` is an implementation of Theorem 1 for the equivalence relation $\equiv$ over CHR states. In this module, CHR states (cf. Definition 1) are represented by Prolog terms of the form `state(G,B,V)`, where `G`, `B`, and `V` are lists representing the goal store, the built-in store and the global variables of the state. CHR constraints are represented by Prolog terms, built-in constraints by the terms `=/2`, `true`, and `false` (which are the only supported built-ins, see Section 5.1) while variables are represented as Prolog variables. The empty goal and built-in stores $\top$ as well as the empty set of variables are represented as empty Prolog lists.

Example 1. The CHR state $\langle\{c(X)\}, X = 1, \{X\}\rangle$ is represented by the term `state([c(X)],[X=1],[X])`.

The predicate `equivalent_states/2` takes two `state` terms representing CHR states σ and σ' as arguments and succeeds if and only if $\sigma \equiv \sigma'$.

Example 2. The goal

```
equivalent_states(state([c(X)],[X=1],[X]), state([c(1)],[X=1],[X]))
```

succeeds while the following goal fails:

```
equivalent_states(state([c(X)],[],[X]), state([c(X)],[X=1],[X]))
```

3.2 Confluence

Based on the module for state equivalence, the module `conflcheck` implements the criterion for confluence from Theorem 2. The confluence checker creates all possible critical pairs of all rules in the program according to Definition 7. It checks them for joinability by posing the two states separately as query to the CHR program and retrieving the two resulting states, which are tested for equivalence by `equivalent_states/2`.

The entire list of critical pairs is created before each individual critical pair is tested for joinability. This allows for filtering out critical pairs which are variants or symmetrical to other critical pairs. The list is also filtered to remove critical pairs whose states are already equivalent. Only the remaining critical pairs are tested for joinability.

The module `conflcheck` defines the predicate `check_confluence/1` which takes the path to a CHR program file as argument and checks this program for confluence. The predicate always succeeds, printing either a message of success or a list of non-joinable critical pairs.

Example 3. Consider the following program simulating destructive assignment, which can also be found in `examples/mem.pl` in the package available at [6]:

```
assign(V,N), cell(V,O) <=> cell(V,N).
```

This program is not confluent as it has two non-joinable critical pairs. The call `check_confluence('examples/mem.pl')` outputs information about the two non-joinable critical pairs as follows (shortened):

```
Checking confluence of CHR program in examples/mem.pl...

The following critical pair is not joinable:
state([cell(A, B), cell(C, D)], [C=A, D=E], [C, D, F, A, E, B])
state([cell(C, F), cell(A, E)], [C=A, D=E], [C, D, F, A, E, B])
...
The following critical pair is not joinable:
state([assign(A, B), cell(C, D)], [C=A, E=F], [C, D, E, A, B, F])
state([assign(C, D), cell(A, B)], [C=A, E=F], [C, D, E, A, B, F])
...
The CHR program in examples/mem.pl is NOT confluent!
2 non-joinable critical pair(s) found!
```

The predicate `check_confluence/3` works the same way but only considers critical pairs of two (not necessarily different) rules in the program. The module `conflcheck` also offers predicates which return the number of non-joinable critical pairs instead of printing them. Those predicates can be used to call the confluence checker from other programs. Details about the mentioned predicates and additionaly ones are given in the manual of the confluence checker [6].

4 Evaluation and Related Work

In this section, we describe the results of applying our confluence checker to programs which have already been analyzed for confluence as well as we compare our implementation to existing confluence checkers for CHR programs.

4.1 Confluence of Union-Find

In addition to unit-tests with classic CHR programs, which are defined in the file `tests.pl` in the package available at [6], we evaluated our confluence checker with the CHR implementations of the union-find algorithm from [5]; on the one hand to test our checkers with programs yielding more critical pairs, on the other hand to compare the results of our confluence checker to a previous confluence analysis from [5].

We were able to confirm the number of critical pairs of the implementations in `ufd_basic.pl`, `ufd_basic1.pl`, and `ufd_rank.pl` for those rules our confluence checker is applicable to.

However, checking the confluence of the parallelized optimal union-find implementation in `ufd_found_compr.pl`, we found differing numbers of non-joinable critical pairs for some rules:

– The rule `findroot1 @ root(A,_) \ find(A,X) <=> found(A,X)` has two non-trivial critical pairs that we both found to be joinable in contrary to one non-joinable critical pair mentioned in [5].
– For the rule `compress @ foundc(C,X) \ A~>B, compr(A,X) <=> A~>C` we found the following four non-joinable critical pairs, in contrary to [5], where only three non-joinable critical pairs were found:

```
(foundc(C,X)  ∧  A~>C  ∧  A~>D,
 foundc(C,X)  ∧  A~>C  ∧  A~>B)

(foundc(C,X)  ∧  foundc(D,E)  ∧  A~>C  ∧  compr(A,E),
 foundc(C,X)  ∧  foundc(D,E)  ∧  A~>D  ∧  compr(A,X))

(foundc(C,X)  ∧  foundc(D,X)  ∧  A~>C,
 foundc(C,X)  ∧  foundc(D,X)  ∧  A~>D)

(foundc(C,X)  ∧  A~>C  ∧  foundc(D,X)  ∧  A~>E,
 foundc(C,X)  ∧  A~>D  ∧  foundc(D,X)  ∧  A~>B)
```

– The rule `linkeq1c @ found(A,X), found(A,Y), link(X,Y) <=> foundc(A,X), foundc(A,Y)` leads to 18 non-joinable critical pairs according to [5]. However, there are only 13 different possibilities to overlap the head constraints of this rule leading to six non-joinable critical pairs.

Using our new tools, we were able to correct some of the numbers of non-joinable critical pairs. We suppose the differing numbers to be due to typos or an error in the existing implementation as it can be easily shown that our above-mentioned numbers are correct. However, despite our findings, the general results presented in [5] remain correct in that the programs are not confluent.

4.2 Related Work

To our knowledge there are two existing implementations of confluence analysis for CHR programs. The tool used to check confluence of the union-find implementations in [5] requires the CHR program to be represented as Prolog facts in the source code of the tool. Another existing confluence checker [2], whose sources are available to the authors, is limited to single-headed simplification rules. Furthermore, the existing tools check joinability of critical pairs based on the notion of variants rather than state equivalence. Our confluence checker overcomes those limitations as it supports multi-headed and simpagation rules and directly parses a CHR source file as input. Also, it checks joinability based on state equivalence, according to Definition 5.

Both existing confluence checkers require manual changes in the source code of the analyzed CHR program in order to process built-in constraints. Our implementation does not need any changing of the source file to process built-in constraints. However, as a trade-off, it is restricted to the built-ins `=/2`, `true`, and `false` as of now (cf. Section 5.1).

5 Conclusion and Future Work

We have presented the first implementation of a program for state equivalence testing and a new confluence checker based on an axiomatic definition of state equivalence. We used a modular design for our checkers to allow for extensions and reuse of our code. We have tested them with unit-tests and existing CHR implementations of the union-find algorithm, re-evaluated the previously published results about their numbers of critical pairs, and compared our confluence checker to existing implementations.

5.1 Future Work

Due to our definition of states which does not provide any means to express the propagation history of the operational semantics ω_t [1], our confluence checker does not work for programs containing propagation rules. Taking critical pairs from propagation rules alongside the propagation history into account is one possible solution for this. Alternatively, the confluence checker can be extended to support persistent constraints [9,4].

To allow arbitrary Prolog predicates as built-in constraints, the checkers for state equivalence and confluence need to be extended to check logical entailment. The current restriction to the built-ins `=/2`, `true`, and `false` arises from the fact that Prolog in general requires the arguments of built-ins to be ground, while CHR states and especially critical pairs can contain unbound variables.

An extension to the notion of confluence is the so-called *observable confluence* [7]. The criterion for observable confluence only considers critical pairs satisfying an invariant. Our confluence checker can be adapted to this notion of confluence: the predicate `process_cps/2` can be used for any kind of modification of critical pairs and thus can be extended to test and alter each critical pair according to a given invariant. Alternatively, the addition of an interactive mode could enable the user to check and modify each critical pair before it is tested for joinability.

In [10], a criterion for operational equivalence of CHR programs, which also relies on joinability, is given. Our implementations of critical pair generation and state equivalence can be adapted and used to implement a checker for operational equivalence.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
2. Abdennadher, S., Frühwirth, T., Meuss, H.: On Confluence of Constraint Handling Rules. In Freuder, E., ed.: Principles and Practice of Constraint Programming, Second International Conference, CP 96. Volume 1118 of Lecture Notes in Computer Science., Springer-Verlag (1996) 1–15
3. Raiser, F., Betz, H., Frühwirth, T.: Equivalence of CHR States Revisited. In Raiser, F., Sneyers, J., eds.: Sixth International Workshop on Constraint Handling Rules (CHR). (2009) 34–48
4. Betz, H., Raiser, F., Frühwirth, T.: A Complete and Terminating Execution Model for Constraint Handling Rules. In: Logic Programming, 26th International Conference, ICLP 2010. (2010) accepted.
5. Frühwirth, T.: Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis. In Gabbrielli, M., Gupta, G., eds.: Logic Programming, 21st International Conference, ICLP 2005. Volume 3668 of Lecture Notes in Computer Science., Springer-Verlag (2005) 113–127
6. Langbein, J.: A State Equivalence and Confluence Checker for CHR, Prolog sources. `http://www.uni-ulm.de/en/in/pm/research/topics/chr/info/downloads.html`
7. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable Confluence for Constraint Handling Rules. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007. Volume 4670 of Lecture Notes in Computer Science., Springer-Verlag (2007) 224–239
8. Abdennadher, S.: Operational Semantics and Confluence of Constraint Propagation Rules. In: Principles and Practice of Constraint Programming, Third International Conference, CP97. Volume 1330 of Lecture Notes in Computer Science., Springer-Verlag (1997) 252–266
9. Betz, H., Raiser, F., Frühwirth, T.: Persistent Constraints in Constraint Handling Rules. In: (Constraint) Logic Programming, 23rd Workshop, WLP 2009, Institute of Computer Science, Potsdam University (2009)
10. Abdennadher, S., Frühwirth, T.: Operational Equivalence of CHR Programs and Constraints. In Jaffar, J., ed.: Principles and Practice of Constraint Programming, Fifth International Conference, CP 99. Volume 1713 of Lecture Notes in Computer Science., Springer-Verlag (1999) 43–57