ulm university universität
uulm

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie

Institut für
Programmiermethodik und
Compilerbau

Extension of the Confluence Checker for CHR-Programs

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Daniel Martin Yaspar Bebber
daniel.bebber@uni-ulm.de

Gutachter:
Prof. Dr. Thom Frühwirth

Betreuer:
M.Sc. Daniel Gall

2015

Version October 28, 2015

© 2015 Daniel Martin Yaspar Bebber

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to
Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.
Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

There are many occasions where it is interesting to know if a program is confluent. At the moment there is a tool to check the confluence of CHR-programs, written by Johannes Langbein. However this checker can only check CHR-programs for confluence that only contain simplification and simpagation rules. For this thesis a new confluence checker is built that supports propagation rules. It is influenced by the program of Johannes Langbein. With a confluence checker that supports this third type of rules the completion algorithm for non-confluent CHR-programs can be implemented. This algorithm tries to add new rules to a program to fulfil the confluence property. This is also done in this work. The algorithm uses the confluence checker to realise two of its four inference rules.

Contents

1 Introduction

The confluence property of a program, using a rule-based language, guarantees that any computation for a goal, a rule in the given program, results in the same final state no matter which of the applicable rules are applied. Confluence as a consequence also ensures that the order of rules in a program and the order of constraints, objects that contain information, in a goal does not matter for the result. There is a decidable, sufficient, and necessary test for confluence for terminating programs that returns the conflicting rule applications [6, p.101]. A disk is in the appendix of this thesis. The content of this disk is described at the end of this work.

1.1 Motivation

There are many cases where knowing if a program can be considered confluent is interesting and important. It is important if you want to run a program on multiple processors parallel or want to find errors in you calculation. When a program is not confluent knowing how to restore the confluence property might be interesting to ensure the possibility of parallel processing or reducing the possibility of errors. Checking if a given program is confluent can be done by hand but is getting very complex with rising complexity of the program. An automatised tool for confluence checks and completion saves time and is making complex programs checkable. Such a tool was written by Johannes Langbein [1], but since this checker does not support all types of CHR-Rules this new checker was implemented.

1.2 Aim

The aim of this work is create a new confluence checker for CHR-programs with influences of the checker made by Johannes Langbein [8] that supports propagation rules. That are rules that add information without removing the constraint that made the rule applicable. The problem with propagation rules is that a list where every already propagated constraint is saved is needed, the so called propagation history. Without it the propagation rules would be able to fire endlessly so the program would not be terminating. The main goal of this work is writing a confluence checker that has full support for the propagation rules.

With a confluence checker that supports propagation rules the completion algorithm mentioned in [6, p. 113] can be implemented. The full aim of this work is to create a program that can perform both - the confluence check and the completion if possible.

This work starts with a short overview over Constraint Handling Rules (CHR) and the principles of confluence and completion in chapter 2. After the basics have been shown the theory and the implementation of the new confluence checker is shown, together with testcases and examples in chapter 3. Since the new checker uses the propagation history it is described in the chapter 4. A confluence checker that supports propagation rules can also try to complete a given non-confluent program, the implementation of the completion algorithm together with all changes to the checker is shown chapter 5. After both implementations have been shown and discussed two chapters that focus on the built-in support, chapter 6 and the output, chapter 7 are given. In chapter 8 the correctness of the new checker is evaluated and (dis-)advantages of the new implementation is shown.

1.3 Related Work

The basis for this work is the book [6] written by Thom Fruewirth. In this book the confluence check and the completion algorithm are described. Both are only described theoretically but with proven algorithms.

One current implementation for this check are [1] by Johannes Langbein a basic confluence checker that supports simplification and simpagation rules. The checker is based on the theory in [6]. The implementation was done in fully in Prolog. The advantage of this checker is that it can check the confluence for simple CHR-Programs. While the disadvantage is that it cannot handle propagation rules and has a limited support for built-ins.

Another implementation is build on the first checker. Frank Richter extended the checker [1] by adding support for more built-ins [10]. The implementation was done in Prolog and CHR. It has all advantages of the first checker and the support for built-ins gives the user the option to check more complex programs. A limitation is that the constraints in the input program cannot have names that are used by the checker itself. Since the name are explicit shown in [10] this limitation is no real problem. The disadvantage of the checker is that it also does not support propagation rules.

2 Constraint Handling Rules

Constraint Handling Rules (CHR) was invented in 1991 by Prof. Dr. Thom Frühwirth [7]. CHR is a high-level programming language, that offers a theoretical formalism related to first-order logic and linear logic, while being a practical programming language based on rules [6, p. xvii].
CHR always needs a host language *H* that provides the data types and predefined constraints. While using CHR the host language is denoted in round brackets and needs to provide at least the constraints *true* and *false*, and syntactic equality and inequality checks. CHR(*H*) denotes CHR embedded in the host language *H*. There are several implementations like CHR(*C*), CHR(*Java*) or CHR(*Haskell*) but CHR(*Prolog*) is the most common one [11].

2.1 Syntax

CHR is a first-order-logic language - so its signature consists of a set of variables *V*, a set of function symbols Σ, and a set of predicate symbols Π. Functions and predicates got a number of *arguments* they take, the so-called *arity*. The arity of a symbol *f* with the arity *n* is specified by the notation *f/n* and we call this a *functor*. If a function symbol has arity zero it is called a *constant*, predicate symbols with arity zero are called *propositions* [6, p. 49].

$Built-in\ constraint:$	C, D	$::= c(t_1, ..., t_n) \mid C \wedge D, n \geq 0$
$CHR\ constraint:$	E, F	$::= e(t_1, ..., t_n) \mid E \wedge F, n \geq 0$
$Goal:$	G, H	$::= C \mid E \mid G \wedge H$
$Simplification\ rule:$	SR	$::= r@E \Leftrightarrow C \mid G$
$Propagation\ rule:$	PR	$::= r@E \Rightarrow C \mid G$
$Simpagation\ rule:$	SPR	$::= r@E1 \setminus E2 \Leftrightarrow C \mid G$
$CHR\ rule:$	R	$::= SR \mid PR \mid SPR$
$CHR\ program:$	P	$::= R_1, ...R_m, m \geq 0$

Figure 2.1: Abstract syntax of CHR programs and rules [6, p. 54]

Definition 1. *A CHR Program (P) consists of a finite set of CHR rules (R). There are three types of rules: simplification (SR), propagation (PR), and simpagation rules(SPR). The structure of these rules can be seen in figure above. A rule can have a name r. Each type of rule has a set of CHR constraints that must not be empty (E) and form the head of the rule, a rule symbol*

($\Rightarrow, \Leftrightarrow$) that specifies the type of rule that is used, a guard that may be empty and consists of built-in constraints (C), those constraints are part of the host language, and a goal that may not be empty and can have built-in constraints as well as CHR constraints (G).

2.2 Semantics

This section describes the operational semantics of CHR. There are two definitions for these, one for the analysis and one for the practical implementation. Only the very abstract and the abstract semantics will be described here, because they are sufficient to explain confluence.

In the following a CHR Program P and a constraint theory CT for the built-in constraints is given. A rule whose head is satisfied by the CHR constraints in the store gets applied and adds all constraints of its goal to the constraint store. When the rule was a simplification rule the head constraints of the rule will be removed from the store while a propagation rule will not remove these . A simpagation rule has both – a head where with constraints that are removed and one where they are kept.

2.2.1 Very Abstract Semantics

The very abstract operational semantics of CHR is given by a nondeterministic state transition system [6, p. 55].

Definition 2 (State)**.** *A state is a conjunction of built-in and CHR constraints. An initial state (initial goal) is an arbitrary state and a final state is one where no more transitions are possible [6, p. 55]*

For transitions the head normal form (HNF) is used. This means, that each argument of a head constraint is a unique variable. A rule can be represented in HNF by replacing each of its head arguments t_i with a new variable V_i and adding the equation $V_i = t_i$ to the guard of the rule. Each transition represents a rule application. In Figure 2.2 the formal definition of the transition relation can be seen. H_1, H_2, C, B and G represent conjunctions of constraints that also can be empty. If a applicable rule is applied the CHR constraints H_1 are kept while the CHR constraints H_2 are removed. The state that results out of this transition also consists the guard C and the body B [6, p. 56]

This transition system is nondeterministic, because there are cases where several rules are applicable but only one can be chosen nondeterministically and this choice cannot be undone (committed choice) [6, p. 56].

Apply

$$(H_1 \wedge H_2 \wedge G) \rightarrow_r (H_1 \wedge C \wedge B \wedge G)$$

if there is an instance with new local variables $\bar{x}$ of a rule named r in P

$$r@H_1 \setminus H_2 \Leftrightarrow C|B$$

and $CT \models \forall(G \rightarrow \exists\bar{x}C)$

Figure 2.2: Transition of the very abstract operational semantics of CHR [6, p. 56]

2.2.2 Abstract Semantics ω_t

The very abstract semantics (Section 2.2.1) got such a high grade of abstraction that they do not care about termination. There are trivial cases for states that do not terminate like transitions with propagation rules of inconsistent built-in constraints. Due to the structure of propagation rules the same rule could be applied again and again since additional built-it constraints cannot invalidate an applicability condition that holds. An example for such a rule would be: $a \Rightarrow b$. With no further definition and an a constraint in the store this rule could fire endlessly. Additional a false formula implies any formula so a inconsistent built-it constraint leads to a applicability condition that holds every time. These issues are addressed by the abstract operational semantics of CHR, that add a distinction between processed and unprocessed constraints this realises that propagation rules cannot be applied more than one time with the same constraints [6, p. 59].

In a failed state any rule is applicable and it can only lead to another failed state. Due to this all failed states are declared as final states to prevent nontermination of failed states. To avoid the repeated application of propagation rules the same constraints can only be applied to a rule once. This information is stored in the propagation history [6, p. 60].

Like in the very abstract semantics, the guard is a conjunction of built-in constraints, but the head and the goal of the rule are now multisets of atomic constraints. Each constraint now has a unique identifier. For a CHR constraint c with the identifier i the notation $c\#i$ is used, such a constraint is called a numbered constraint. The functions $chr(c\#i) = c$ and $id(c\#i) = i$ are used for numbered CHR constraints and extended to sequences and sets of numbered CHR constraints in the obvious way [6, p. 60]. One new possibility that these identifiers offer is the identification of constraints that were already used on a propagation rule.

Definition 3. *[6, p. 60] A ω_t state is a tuple of the form $\langle G, S, B, T\rangle_n^V$.*

- *The goal (store) G is a multiset of constraints to be processed.*
- *The CHR (constraint) store S is a set of numbered CHR constraints that can be matched with rules in a given program P.*
- *The built-in (constraint) store B is a conjunction of built-in constraints that has been passed to the built-in constraint solver.*
- *The propagation history T is a set of tuples (r, I) where r is the name of a rule and I is the*

sequence of the identifiers of the constraints that matched the head constraints of r.

- *The counter n represents the next free integer that can be used as an identifier for a CHR constraint.*
- *The sequence V contains the variables of the initial goal.*

Definition 4 (Failed states and final states)**.** *[6, p. 61] Given an initial goal (query, problem, call) G with variables V, the initial state is $\langle G, S, B, T\rangle_1^V$.*

A state $\langle G, S, B, T\rangle_n^V$ with inconsistent built-in constraints $(CT \models \neg\exists B)$ is called failed. A state with consistent built-in constraints and empty goal store $(G = \emptyset)$ is called successful. The remaining kinds of states have no special name. A final state $\langle G, S, B, T\rangle_n^V$, its conditional or qualified answer (solution, result) is the conjunction $\exists\bar{y}(chr(S) \wedge B)$, where $\bar{y}$ are the variables not in V.

In Figure 2.3 the transition rules for the abstract operational semantics ω_t are shown. With the solve transition the built-in solver adds a built-in constraint from the goal G to the built-in constraint store B. This results in a simplification of the built-in store , but in the worst case it is just the original conjunction of the new constraint with the old built-in store. The introduce transition adds a new CHR constraint the the CHR store S and assigns an identifier to that constraint (the next free integer $_n$). In the apply transition a rule from the program P gets picked and applied (fired, executed). The criterion for a picked rule is that there are constraints in the CHR constraint store S that match the head of the rule and that the guard g of the rule is logically implied by the built-in store B under the matching [6, p. 61].

Solve

$$\langle \{c\} \uplus G, S, B, T\rangle_n \mapsto_{solve} \langle G, S, B', T\rangle_n$$

where c is a built-in constraint and $CT \models \forall(c \wedge B) \leftrightarrow B')$.

Introduce

$$\langle \{c\} \uplus G, S, B, T\rangle_n \mapsto_{introduce} \langle G, \{c\#n\} \cup S, B, T\rangle_{(n+1)}$$

where c is a CHR constraint.

Apply

$$\langle G, H_1 \cup H_2 \cup S, B, T\rangle_n \mapsto_{apply\ r}$$
$$\langle C \uplus G, H_1 \cup S, chr(H_1) = H_1' \wedge chr(H_2) = H_2' \wedge g \wedge B, T \cup \{r, id(H_1) + id(H_2))\}\rangle_n$$

where there is a fresh variant of a rule named r in P with variables $\bar{x}$ of the form

$$r@H_1' \backslash H_2' \Leftrightarrow g|C$$

where $CT \models \exists(B) \wedge \forall(B \rightarrow \exists\bar{x}(chr(H_1) = H_1' \wedge chr(H_2) = H_2' \wedge g))$ and

$$(r, id(H_1) + id(H_2) \notin T.$$

Figure 2.3: Transition of the abstract operational semantics ω_t [6, p. 63]

2.3 State Equivalence

This section uses the following definition for states that can be found in [9]:

Definition 5 (State)**.** *A CHR state σ is a tuple $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ where the goal $\mathbb{G}$ is a multiset of CHR constraints, the built-in constraint store $\mathbb{B}$ is a conjunction of built-in constraints and the variables $\mathbb{V}$ are a set of global variables. A variable that is an element of $(\mathbb{G} \cup \mathbb{B})$ and not an element of $\mathbb{V}$ is called a local variable. If a variable is only an element of $\mathbb{B}$ and not an element of $(\mathbb{G} \cup \mathbb{V})$ it is called a strict local variable.*

With these definitions it would be possible to use the logical equivalence when talking about state equivalence, but since all failed states count as equivalent and CHR constraints got a multiset character, a more strict notion of equivalence is needed. The basic idea of state equivalence in CHR is, that the same rules should be applicable to those states [6, p. 71].

Definition 6 (State Equivalence)**.** *Equivalence between CHR states is the smallest equivalence relation $\equiv$ over CHR states that satisfies the following conditions [9]:*

1. (Equality as Substitution)

$$\langle \mathbb{G}, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle \equiv \langle [x/t], x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$$

2. (Transformation of the Constraint Store)

If $CT \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ where $\bar{s}$,$\bar{s}'$ are strict 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-Occurring 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)

$$\langle \mathbb{G}, \bot, \mathbb{V} \rangle \equiv \langle \mathbb{G}', \bot, \mathbb{V} \rangle$$

A necessary and sufficient criterion for deciding state equivalence is given by the following Thorem 1 [9]:

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

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

Since this theorem only applies if the set of global variables is unchanged and the local variables are renamed apart, it actually decides a smaller relation than $\equiv$. But due to the fact that we can rename local variables apart and that Definition 6.3 allows us to adjust the set of global variables, those restrictions do not result in a problem [9].

2.4 Confluence

The confluence property of a program guarantees the same final state for any computation of a goal independent of which applicable rules are fired. If a program is confluent the order of rules and constraints in a goal do not matter - the result is the same [6, p. 101]. A CHR program is well-behaved if it is terminating and confluent [6, p. 102]. Confluence was presented first in [4].

Definition 7 (Confluence)**.** *A CHR program is confluent if for all states* S, S_1, S_2

$$If\ S \mapsto^* S_1, S \mapsto^* S_2\ then\ S_1\ and\ S_2\ are\ joinable.$$

2.5 Minimal States

When a program is analysed, a big problem results in the fact, that there are an infinite amount of possible states. So it is necessary to reduce the amount of states to a level that can be analysed, the so-called minimal states. Every state has this minimal, most general, state that allows it to fire [6, p. 101].

Definition 8 (Minimal State)**.** *The minimal state of a rule is the conjunction of the head and the guard of a rule [6, p. 101].*

If only one constraint from such a minimal state gets removed, the rule would be no longer able to fire. Adding constraints cannot prevent the possibility of the rule to fire. The conclusion of this is that every state that allows a rule to fire contain at least the minimal state of the rule [6, p. 101].

2.6 Joinability

The property joinability describes two states that result in equal states. This is an important property for the confluence check.

Definition 9 (Joinability)**.** *Two states* S_1 *and* S_2 *are joinable if there exist states* S_1',S_2' *such that* $S_1 \rightarrow^* S_1'$ *and* $S_2 \rightarrow^* S_2'$ *and* $S_1' \equiv S_2'$ *[6, p. 102].*

2.7 Test for confluence

In general there is an infinite amount of possible states, since it is possible to execute a program with an infinite amount of different constraints. For a terminating program it is possible to limit these states for the joinability test with the finite number of most general states, the so-called overlaps of the rules. These overlaps are states where more than one rule is able to fire. By

merging the minimal states of two rules where at least one constraint in each rule is the same these two rules are overlapped. The two states resulting from the application of these both rules are called a critical pair. If any critical pair is not joinable, the program itself is not confluent. The most important property of a program to check this is that the program is terminating [6, p. 102].

Definition 10. *Let R_1 be a simplification or simpagation rule and R_2 be a rule, whose variables have been renamed apart. Let $H_i \wedge A_i$ be a conjunction of the head constraints C_i be the guard and B_i be the body of rule $R_i (i = 1, 2)$. Then a (nontrivial) overlap (critical ancestor state) S of rules R_1 and R_2 is*

$$S = (H_1 \wedge A_1 \wedge H_2 \wedge (A_1 = A_2) \wedge C_1 \wedge C_2),$$

provided A_1 and A_2 are nonempty conjunctions and the built-in constraints are satisfiable,

$CT \models \exists((A_1 = A_2) \wedge C_1 \wedge C_2)$.

Let $S_1 = (B_1 \wedge H_2 \wedge (A_1 = A_2) \wedge C_1 \wedge C_2)$ and $S_2 = H_1 \wedge B_2 \wedge (A_1 = A_2) \wedge C_1 \wedge C_2)$. Then the tuple (S_1, S_2) is a critical pair (c.p.) of R_1 and R_2.

A critical pair (S_1, S_2) is joinable, if S_1 and S_2 are joinable [6, p. 103].

For terminating CHR programs, a decidable, sufficient and necessary condition for confluence is given by the following theorem [6, p. 104]:

Theorem 2. *A terminating CHR program is confluent iff al its critical pairs are joinable [6, p.104] [2] [5].*

2.7.1 Propagation history

A program to check the confluence of CHR programs that only contain simplification and/or simpagation rules was written by Johannes Langbein in 2010 [8]. To build an enhanced confluence checker that supports propagation rules, the propagation history H must be implemented. H contains a list of all CHR constraints that got applied to a propagation rule. When a propagation history is being used the constraints have to be altered since an ID is added to them and the rules of the program are also getting an unique identifier. The propagation history is the core element to prevent the program from infinite looping since the same constraint would get applied to the same rule again and again [6, p. 61].

Let h be a part of H, then h has the following structure:

h = (C^{idc}, R^{idr}) where C is a list of the used constraints in the rule R.
The constraints C get a unique ID idc and the rules R, too (idr).

2.7.2 Examples

In this section some example CHR programs are given which are tested for confluence. Those examples are taken from [6]. A CHR program that only contains propagation rules or single headed simplification rules with no overlap are trivial cases that are obviously confluent [6, p. 105].

Example 1. *The following coin-throw example is an abstraction of the example given in [6]:*

$$coin(throw) \Leftrightarrow head.$$
$$coin(throw) \Leftrightarrow tail.$$

The only overlap is (simplified for readability):

$$coin(throw)$$

and it leads to the critical pair:

$$(head, tail)$$

These two states are final and different. Thus, they are not joinable.

Example 2. *A simple example with two simplification rules:*

$$p \Leftrightarrow q.$$
$$p \Leftrightarrow false.$$

This program got a single overlap $(q, false)$ *that consists of final states that are not joinable. Hence the program is not confluent.*

Example 3. *This examples consists of a single rule.*

$$p(X) \wedge q(Y) \Leftrightarrow true.$$

The program is not confluent, since the only rule of the program got an overlap with itself, as this example input shows:

$$p(X) \wedge q(Y1) \wedge q(Y2)$$

This overlap leads to the critical pair $(q(Y1), q(Y2))$*, where* $Y1$ *and* $Y2$ *are different variables. Since there is no specification in CHR which constraint has to be taken first this case is always possible. A case where this nonjoinability does not arise would be:*

$$p(X) \wedge q(Y) \Leftrightarrow X = Y \mid true$$

Example 4. *The following example contains a propagation rule:*

$$\begin{aligned} r1 \quad @ \quad & p \Rightarrow q. \\ r2 \quad @ \quad & r \wedge q \Leftrightarrow true. \\ r3 \quad @ \quad & r \wedge p \wedge q \Leftrightarrow s. \\ r4 \quad @ \quad & s \Leftrightarrow p \wedge q. \end{aligned}$$

Which has the overlap:

$$r \wedge p \wedge q$$

Figure 2.4 shows that the propagation history is important. Without it the program would always end in the final state $p \wedge q$ (as seen in (1)). However with the propagation history for $r1$ it results in the nonjoinable critical pair $(p, p \wedge q)$ (as seen in (2)).

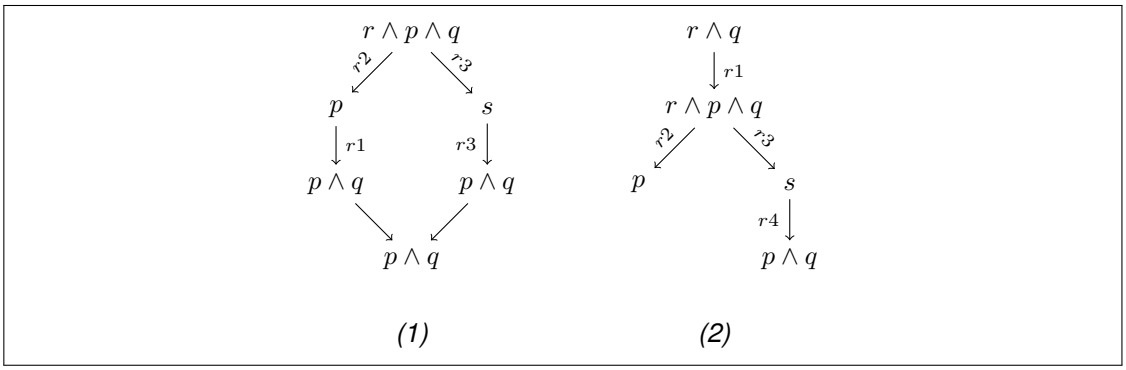

Figure 2.4: Violated confluence in lager state [6, p. 109].

2.8 Completion

Completion for CHR is a process of adding rules to a nonconfluent program until it becomes confluent [6, p. 112]. For the generation of these rules the critical pairs of the given program P are needed. The completion algorithm for CHR generally needs more than one rule to make critical pairs joinable (and thus solving the nonconfluence). The generation of the new rules is not always possible. The generation of new rules may also introduce new critical pairs that need to be checked again for confluence. Since this check can lead to new nonjoinable critical pairs that have to be completed the algorithm may not be terminating. Completion was introduced in [3].

2.8.1 Completion algorithm

The completion algorithm is specified by a set of inference rules [6, p. 112]. These four rules are called Simplification, Deletion, Orientation and Introduction. In addition there is the function

$orient$ which tries to generate a simplification and a propagation rule from a given critical pair. The function is partial and does not apply if rules cannot be generated.

Definition 11. *[6, p. 112] Let $\gg$ be a termination order. Let $(E_i \wedge C_i, E_j \wedge E_j)(i, j \in \{1, 2\})$ be a nonjoinable critical pair, where E_i, E_j are CHR constraints and C_i, C_j are built-in constraints. The partial function orient $\gg$ applies to the set of the two states in the critical pair $\{E_1 \wedge C_1, E_2 \wedge C_2\}$ if $E_1 \wedge C_1 \gg E_2 \wedge C_2$ if E_1 is a nonempty conjunction, and if E_2 is a nonempty conjunction of $CT \models C_2 \rightarrow C_1$. It returns a set of rules:*

$$\{E_1 \Leftrightarrow C_1 | E_2 \wedge C_2, E_2 \Rightarrow C_2 | C_1\}$$

where the propagation rule is generated only if $CT \not\models C_2 \rightarrow C_1$.

The task of the propagation rule is to ensure that the built-ins of both states in the critical pair are enforced. The function does not add redundant propagation rules $(CT \models C_2 \rightarrow C_1)$ and the conditions are carefully chosen so that it does not apply if the two states in the critical pair cannot be ordered by $\gg$ or if a rule with an empty head would result [6, p. 113].

Simplification:	If $S_1 \mapsto S_1'$ then $(C \cup \{\{S_1, S_2\}\}, P) \longmapsto (C \cup \{\{S_1', S_2\}\}, P)$
Deletion:	If S_1 and S_2 are joinable then $(C \cup \{\{S_1, S_2\}\}, P) \longmapsto (C, P)$
Orientation:	If $orient \gg (\{S_1, S_2\}) = R$ then $(C \cup \{\{S_1, S_1\}\}, P) \longmapsto (C, P \cup R)$
Introduction:	If (S_1, S_2) is a critical pair of P not in C then $(C, P) \longmapsto (C \cup \{\{S_1, S_2\}\}, P)$

Figure 2.5: [6, p. 113] Inference rules of completion.

At the beginning of completion there is a given program P and its set of nonjoinable critical pairs S, (S, P). At this point the inference rule can be applied in the order given in figure 2.5 until exhaustion or failure. The rule Simplification replaces a state in a critical pair by its successor state (a rule application). When this rule is applied to exhaustion we only work with final states [6, p. 113]. The Deletion rule removes joinable critical pairs, since they do not hurt the confluence property we do not need them. The Orientation rule removes a critical pair from C and adds new rules to P as computed by $orient$. If $orient$ cannot be applied this inference is not possible. This rule gets applied once. After these first three steps are finished all new critical pairs between the new rules and the old ones are computed by the Introduction rule.

The final successful state of the completion algorithm is $(\emptyset, P')$, otherwise it has failed. Completion fails if a nonjoinable critical pair cannot be oriented. In such a critical pair the states cannot be ordered or they consist of different built-in constraints only [6, p. 113]. Completion may also not terminate, this is the case when new rules get produced endless. Sometimes a different termination order can lead to termination of the completion algorithm.

2.8.2 Examples

In the following examples we use a simple genetic termination order where built-in constraints are the smallest, atomic CHR constraints are ordered by their symbols, and conjunctions of CHR constraints are larger than their conjuncts. These examples are taken from: [6, p.114].

Example 5. *A simple example with two simplification rules:*

$$p \Leftrightarrow q.$$
$$p \Leftrightarrow false.$$

The program is not confluent since p *leads to the critical pair* $(q, false)$*. Simplification and Deletion do not apply. Orientation adds the rule* $q \Leftrightarrow false$ *via the function* $orient_{\gg}$ *to the program. A propagation rule is not produced since* $CT \models false \rightarrow true$*. Introduction does not apply since the new rule has no overlap with the old ones. So the completion terminates successfully with the new program:*

$$p \Leftrightarrow q.$$
$$p \Leftrightarrow false.$$
$$q \Leftrightarrow false$$

Example 6. *Given the following CHR program where* $\geq$ *and* $\leq$ *are built-in constraints, let* $p \gg r \gg q$ *in the generic termination order:*

$$r1 \quad @ \quad p(X,Y) \Leftrightarrow X \geq Y \wedge q(X,Y).$$
$$r2 \quad @ \quad p(X,Y) \Leftrightarrow X \leq Y \wedge r(X,Y).$$

The program is no confluent since the critical pair stemming from $r1$ *and* $r2$

$$(X \geq Y \wedge q(X,Y), X \leq Y r(X,Y))$$

is not joinable. Completion inserts two new rules:

$$r3 \quad @ \quad r(X,Y) \Leftrightarrow X \leq Y | q(X,Y) \wedge X \geq Y.$$
$$r4 \quad @ \quad q(X,Y) \Rightarrow X \geq Y | X \leq Y.$$

The computations show that it is necessary to add the propagation rule $r4$ *to the program. Consider the query* $p(X,Y)$*. A computation using rule* $r2$ *is:*

$$\begin{aligned} & p(X,Y) \\ \mapsto_{Apply\ r2} \quad & r(X,Y) \wedge X \leq Y \\ \mapsto_{Apply\ r3} \quad & q(X,Y) \wedge X = Y \\ \mapsto_{Apply\ r4} \quad & q(X,Y) \wedge X = Y \end{aligned}$$

The application of the propagation rule $r4$ *does not change the answer. However the computation using* $r1$ *needs* $r4$ *to result in the same answer:*

$$\begin{array}{ll} & p(X,Y) \\ \mapsto_{Apply\ r1} & q(X,Y) \wedge X \geq Y \\ \mapsto_{Apply\ r4} & q(X,Y) \wedge X = Y \end{array}$$

Example 7. *In this last exampled the completion fails because a critical pair cannot be oriented.*

$$\begin{array}{lcl} r1 & @ & p(X,Y) \Leftrightarrow X \geq Y \wedge q(X,Y). \\ r2 & @ & q(X,Y) \Leftrightarrow X \leq Y \wedge q(Y,X). \end{array}$$

the program is not confluent since the critical pair stemming from $r1$ *and* $r2$

$$(X \geq Y \wedge q(X,Y), X \leq Y \wedge q(Y,X))$$

is not joinable. However there is no termination order that can order this critical pair, as can be seen from the rule that would result from function $orient$*:*

$$ro \quad @ \quad q(Y,X) \Leftrightarrow X \leq Y | q(X,Y) \wedge X \geq Y.$$

3 The Confluence Checker for CHR Programs

The confluence checker that is described in this section is built from scratch. The checker by Johannes Langbein [8] was not extended because there were too many changes that were needed to fulfil the requirements for the propagation history and the completion algorithm. The new confluence checker uses parts of the old checker.

The confluence checker for CHR programs checks if a program is confluent. For this it has to check if the minimal states of each overlap are joinable. Overlaps are created by a rule pair that does not consist out of two propagation rules where the two rules got atleast one head constraint in common. A rule can overlap with itself (like in example 15). The confluence checker in its current state can detects if a program is confluent or not if the program only contains simplification and simpagation rules and a set of built-ins. The new confluence check that is described in this thesis also supports propagation rules and a simple set of built-ins. The process of parsing the input program is mostly based on exercises of the lecture "Rulebased Programming" at the Ulm University. If code from [1] has been copied this is mentioned at the code fragments.

3.1 Theoretical Concept

The new confluence checker for CHR programs (CC^p) is based on the theoretical concept mentioned in [6, p. 101-105]. First all overlaps between all rules of the program are searched. From these overlaps all minimal states are generated. The minimal states are a conjunction of the head and the guard of two overlapping rules, the minimal amount of constraints needed to fire both rules. If the guards of two rules contradict, CC^p detects that these rules cannot result into a state of non-confluence.

For each pair of overlapping rules CC^p generates a confluence test. The minimal states of these overlaps are used to generate a constraint store for the calculation. The test of an overlap is finished when no rule can be applied any more. After this state is reached the final states of the overlap are checked for joinability. If the two final states are joinable the overlap does not result into a critical pair, if not the program cannot be confluent.

Since CC^p cannot check all possible permutations of rules and constraints a more simple but still correct method is used. The correctness is ensured by checking only final states. If these final states are identical the overlap is locally confluent. If not a critical non-joinable pair is found.

The reason why this method is correct is because a program that is local confluent in every overlap is confluent. The downside of this method is that the root of the non-confluence of a program is harder to find in some programs. An example for this is the following program:

```
r1 @ a <=> b.
r2 @ a <=> c.
r3 @ b <=> c.
r4 @ b <=> d.
```

The program is obviously not confluent. In the figure below it is shown why:

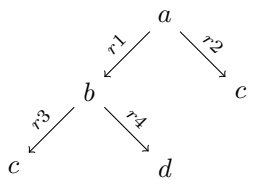

Figure 3.1: Local and global confluence on an example

The CC^p and the checker checker will not find the critical pair that is created by the starting state `(a)` that results into the critical pair `((c),(d))`. They only find the critical pair `((c),(d))` that results from the starting state `(b)`. Both checkers do not check all permutation of the rules and thus not find the critical pair since rule 3 will fire at both checkers and add a `c` to the constraint store. This leads to the joinable state pair `((c),(c))`. They still both give the correct result that the program is not confluent because the local confluence of rule 3 and 4 is not satisfied. The reason for this is that both checkers run in a Prolog implementation. This implementation is working with the refined semantics. In its calculation every rule is applied by its order in the sourcecode. This execution order cannot be influenced.

If the order of rule 3 and 4 are switched both checkers will detect the second critical pair between rule 1 and 2 because the checkers will fire rule 1 and rule 4 when checking the overlap of rule 1 and 2. This leads to the non-joinable state pair `((c),(d))`. To find this critical pair the checkers would need to check all permutations of the rules in the program. This is no problem with small programs but then the checker would not be scalable thus only final states are compared because the global non-confluence is found any time.

3.1.1 Propagation Rules and History

Propagation rules require a propagation history. The theoretically concept of the propagation history is shown after the rule transformation has been shown. For using propagation rules and the history the input program needs to be transformed. Each CHR-Constraint of the input program needs to be altered by increasing its arity by one. This new argument is used for the unique identifier of the constraint. This identifier is needed so that a constraint does not fire

a propagation rule endlessly. In the implementation of the CC^p each rule is parsed and the representation of the constraints in the rules also reflects the increased arity. The description of all CHR-Constraints at the beginning of a program is also altered to reflect this. This method is a explicit implementation of the abstract semantics mentioned in chapter 2.2.2. The identifier that is normally hidden when calling a CHR-Program is explicit given after the transformation. The actual implementation transforms with the following method:

Definition 12 (Program-Transformation)**.** *Let P be a program that is checked for confluence. Let c_n be a constraint that is used in P and c_n#ID an unique identifier that is bound to the constraint. Then the ID of c_n is added as a new argument to the constraint. So a constraint* `c(A1,A2,...,AN)` *is transformed into* `c(A1,A2,...,AN,cID)`*. The constraint definition at the beginning of the program (*`:- chr_constraint` c_1`/N1,` c_2`/N2, ...,` c_n`/NN.`*) is transformed into* `:- chr_constraint` c_1`/N1+1,` c_2`/N2+1, ...,` c_n`/NN+1.`

A rule in the program that has the following structure:

c_1`(A1,...,AN) ==>` c_2`(B1,...,BM).`

Is transformed into the rule:

c_1`(A1,...,AN,`c_1`#ID) ==>` c_2`(B1,...,BM,`c_2`#ID).`

The transformed program is equivalent to the input program since all parts of the CC^p treat the constraints like there would be no identifier. The only situation where the identifier is evaluated is when a propagation rule is fired. In every other rule type the last argument of the constraint is ignored. This guarantees that the transformed program behaves like the original input program and thus checking the confluence correctly.

The correct handling of propagation rules is done by creating a history constraint that contains all propagated rule heads and the calculation identifier. To this constraint all heads that were used in a propagation rule are saved. When the checker tries to apply a propagation rule the history constraint of the current calculation is checked whether is was already used with this rule or not. This method guarantees that rules where the head contains a constraint multiple times like `a, a ⇒ b` is fired two times when the store contains `(a,a)`.

Definition 13 (Possible executions on a propagation rule with a constant set of constraints)**.** *Let* PR *be a propagation rules, (c_1,...,c_n) constraints that are part of the head of* PR *and (#c_1,...,#c_n) be the amount of times the given constraint is contained in the head. Then (#c_1! · #c_2! · ... · #c_n!) is the amount of possible executions when every constraint of the head is given in the constraint store.*

The history constraint saves the possible amount of applications. A counter that is part of each element in the list that is inside the history constraint is increased when the same set of constraints is applied to the same propagation rule again. When the maximum amount of applications is reached the rule cannot fire any more with this set of constraints. This guarantees that

a propagation rule will not loop infinite when the rule is not constructed to do so. A rule like `a, a ⇒ a` will still create an infinite loop since the new `a` constraint allows the rule to fire again. To ensure that the right constraints are checked for their occurrence in the propagation history the unique identifiers in the last argument of the constraint are used. When other rule types than propagation rules are evaluated the history constraint is also in the head of the given and is added to the constraint store again. This is done because the calculation starts as soon as the history constraint is inside the store. Without using it in the heads of the rules for simplification and simpagation rules the calculation starts before the history is inserted. Then the correct flow of the calculation would not be guaranteed.

Definition 14 (Rule Application)**.** *When rule application is simulated inside the CC^p every rule application needs a rule, a store and a history constraint. A rule that has the following structure inside the input program:*

```
a <=> b.
```

Is simulated by the following structure:

```
Rule \ Store_Before, History <=> Check_Store |
                Store_After, History.
```

The concrete implementation is shown in chapter 3.3.3. Simpagation rules are simulated with the same structure.
A propagation rule inside the input program like:

```
a ==> b
```

Is simulated by the following structure:

```
Rule \ Store_Before, History_Before <=> Check_Store, Check_History |
                    Store_After, History_After.
```

In the first example only the store is altered while the history remains untouched. In the second example the store and the history are altered before they are added to the store.

The advantage of this handling is that every type of rule can only fire when the history constraint is part of the constraint store. This ensures that the rules are fired in the order that is wanted.

With the transformation and the usage of the history the correct handling of propagation rules is ensured. The implementation for the transformation is shown in chapter 3.3.1, the concrete one for the propagation history in chapter 4.

3.2 Overview and Requirements

The CC^p works like a pipeline. The concept is shown in figure 3.2. The bold typed parts are the modules that are part of the confluence check itself.

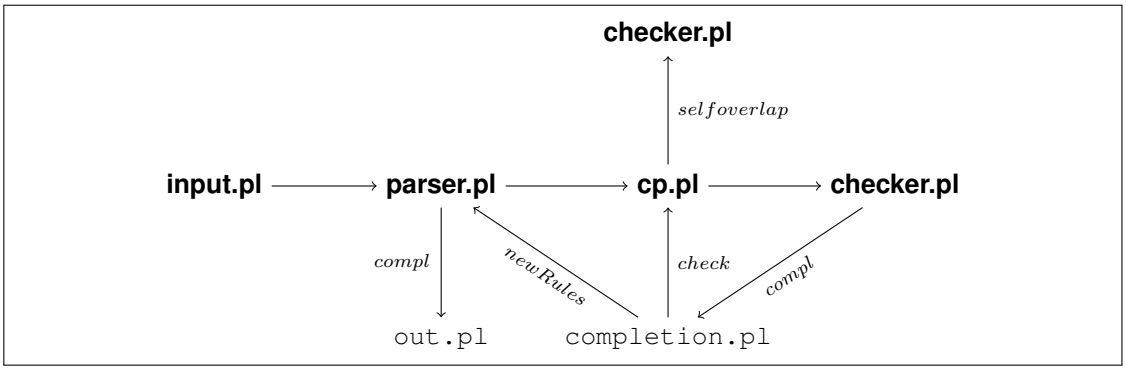

Figure 3.2: The program flow of the CC^p and its output.

Each part of the implementation has its own specialisation. This is mentioned in the name of each of the following section.

When no overlap is found the program stops at the `cp.pl` module and prints it out on console. The paths that contain *compl* are only chosen while completing a program.

The CC^p has to fulfil the following requirements to work correctly:

- read all rules R of the input-program P and transform them.
- check R for (self-)overlaps.
- support the propagation history.
- return the result to the user.

3.3 Checking Confluence

Let P be a syntactically correct and terminating CHR-program then P can be checked for confluence by the CC^p. The main difference between this work and older works like the one of Johannes Langbein [1] is that this checker supports propagation rules. With this ability it can check more programs for confluence than the old one.

Propagation rules are often used to extract additional information from already given constraints. The transitivity relation on orders are an example for this. To implement the relation in CHR the `leq/2` (less or equal) constraint is used:

Example 8. *Transitivity*

```
leq(a,b), leq(b,c) ==> leq(a,c).
```

3.3.1 Parsing P: `parser.pl`

To start the confluence check the predicate `confluence(File_Path)` is called. It is part of the `parser.pl` module. In the first step the parser will read the given input program P and filter out all modules, CHR-constraints and rules. These three parts of P will be store in the constraint `translate/1` where each of the constraints represents one line of P.

The `translate/1` constraint can have different structures:

- `translate(:- use_module(M))`
- `translate(:- chr_constraint C)`
- `translate(Propagation Rule)`
- `translate(Simplification Rule)`
- `translate(Simpagation Rule)`

For the confluence check we need a list of all rules of the program. To create this list all `translate(CHR Rule)` constraints are transformed into another format:

Listing 3.1: Transformation of simplification and simpagation rules with a guard.

```
1 translate(LHS <=> GRD | RHS), rule_list(L) <=>
2                           flatten(LHS <=> GRD | RHS,Out),
3                           Out = [K,R,G,RH],
4                           rule_list([(K,R,'<=>',G,RH)|L]),
5                           chrl(K,R,'<=>',G,RH).
```

The flatten constraint ensures that the constraints and built-ins are represented in a flat list. It returns a list with four elements: The kept head (K), the removed head (R), the guard (G) and the body (RH). The translated rule is added to the `rule_list/1` constraint. In the process of flattening the arity of each constraint is increased by one. This extra argument on each constraint is used to store the unique identifier that is used in the propagation history. The `chrl/1` constraint is used to transform the rules into a structure that is used to write it into the output file when the completion is called. They represent a CHR-Rule.

Listing 3.2: Transformation of simplification rules and simpagation without a guard.

```
1 translate(LHS <=> RHS), rule_list(L) <=>
2                           flatten(LHS <=> RHS,Out),
3                           Out = [K,R,_,RH],
4                           rule_list([(K,R,'<=>',[],RH)|L]),
5                           chrl(K,R,'<=>',[],RH).
```

These checks cover simplification and simpagation rules since the only difference between them is that simpagation rules got a kept head while simplification rules do not. When the rules are used in later steps of the CC^p we can distinguish between the two rule types by these properties.

Listing 3.3: Transformation of propagation rules with a guard.

```
translate(LHS ==> GRD | RHS), rule_list(L) <=>
                          flatten(LHS ==> GRD | RHS,Out),
                          Out = [K,_,G,RH],
                          rule_list([(K,[],'==>',G,RH)|L]),
                          chrl(K,R,'==>',G,RH).
```

The rules for transforming a rule with a guard come first since the one for no guard also applies to the `translate/1` constraint. With this order it is guaranteed that the guards are handled correct.

Listing 3.4: Transformation of propagation rules without a guard.

```
translate(LHS ==> RHS), rule_list(L) <=>
                          flatten(LHS ==> RHS,Out),
                          Out = [K,_,_,RH],
                          rule_list([(K,[],'==>',[],RH)|L]),
                          chrl(K,R,'==>',[],RH).
```

An initial `rule_list/1` with an empty list (`rule_list([])`) is generated at the start of the confluence check. The `translate(:- use_module(M))` constraints are only replaced by `module/1` constraints that contain all used modules of P. To increase the arity in the constraint definition at the start of P the `translate(:- chr_constraint C)` constraint is used together with the `inc_arity/2` predicate:

Listing 3.5: Increase of the arity in the `chr_constraint` definition.

```
translate(:- chr_constraint CL) <=> flattenOH(CL,Out),
                                    inc_arity(Out,NL),
                                    chr_constraints(NL),
                                    ccs(NL).

inc_arity([],_).
inc_arity([X|Xs],NL) :- X =.. [_,N,A],
                        A1 is A+1,
                        append([N/A1],L2,NL),
                        inc_arity(Xs,L2).
```

After all rules were parsed and their arity has been increased the job of `parser.pl` is done. The `rule_list/1` constraint is simplified into a `cleanup/0` constraint that removes all remaining constraints from the store and itself after no other constraint is left and start the next step of the CC^p – the search for overlaps – by calling the predicate `critical_pairs/2` of the module `cp.pl`.

The `parser.pl` transforms P into a different structure like this:

Listing 3.6: Transformation of P.

```
:- use_module(library(chr)).
:- chr_constraint p/0, q/0.

p <=> q.
p <=> false.

Into:

:- use_module(library(chr)).
:- chr_constraint p/1, q/1.

p(_) <=> q(_).
p(_) <=> false.
```

The new argument in the constraints is used to store the unique identifier that is used for the propagation history. The definition of constraints at the beginning of the program is also increased because it is used in later steps to distinguish between built-ins and CHR-Constraints.

3.3.2 Finding overlaps: `cp.pl`

The `cp.pl` module is searching for all overlaps of P. First `cp.pl` searches for possible self-overlaps:

Listing 3.7: Predicates for the overlap check.

```
overlapping([],_).
overlapping([X|XL],RL) :-
        self_overlap(X,RL), overlapping(XL,RL).
```

Listing 3.8: Predicates for the self-overlap check.

```
self_overlap((KH,RH,RS,G,RHS),RL) :-
        append(KH,RH,LHS),
        permutation(LHS,PLHS),
        not(check_overlap(PLHS)),
        overlap_cp((KH,RH,RS,G,RHS),RL).

self_overlap((KH,RH,RS,G,RHS),RL) :-
        append(KH,RH,LHS), check_overlap(LHS).
```

In the process of finding self-overlaps the kept and the removed head of a given rule need to be analysed. The permutation predicate is used because all orders of the CHR constraints need to be checked. Since even a rule with two head constraints that are different and got an arity greater than zero can lead to non-confluence (like in example 3 in chapter 2.7.2) self-overlaps are found often. If there is a self-overlap the predicate adds a `overlap_cp/2` constraint to the CHR-constraint store that contains the self-overlapping rule and the rule list of P. A case where we would not have to check the self-overlap for confluence would be:

```
a(X), a(X) <=> true.
```

In this example the final state does not depend on the order of usage of the CHR-constraints since only a pair of equal constraints are simplified so rules like that cannot lead to critical self-overlaps.

To check the self-overlaps the `check_overlap/1` predicate shown below is used. It checks if the self-overlap leads into a critical pair or not. It fails if the overlap is critical like seen in the second `self_overlap/2` predicate in the figure above. In the following listing the code for `check_overlap/1` is shown:

Listing 3.9: First predicates for checking critical self-overlaps.

```
1 check_overlap([]).
2 check_overlap([X]).
3 check_overlap([X|XL]) :-
4         X =.. XF,
5         length(XF,2),
6         check_overlap(XL).
```

Listing 3.10: Predicates for checking critical self-overlaps.

```
1 check_overlap([X,Y|XL]) :-
2         X =.. XF, reverse(XF,XFR),
3         XFR = [_|XFRL], reverse(XFRL,XFRLR),
4         XN =.. XFRLR,
5         Y =.. YF, reverse(YF,YFR),
6         YFR = [_|YFRL], reverse(YFRL,YFRLR),
7         YN =.. YFRLR, XN == YN,
8         check_overlap([X|XL]), check_overlap([Y|XL]).
```

The `check_overlap/1` predicate first checks if the CHR constraint used in the given rule has variables. If not `length(XF,2)` will succeed because the first element of `XF` is the name of the constraint and the second is the variable that will contain the unique identifier in later steps of the calculation. In the second non-trivial case the first two elements of the list are checked for

equality if the rule contains only equal constraints (with equal variables) it cannot result into a critical pair.

The program `cp.pl` also checks all rules of the program if they got overlaps with other rules. To check this a pair for every rule with another is generated by the `create_pairs(Rule_List)/1` predicate. It creates $\sum_{m=0}^{n-1} m$ pairs where n is the amount of rules in P.

Listing 3.11: Predicate for creating pairs of rules.

```
create_pairs([X,Y|XL]) :-
        pair(X,Y),
        create_pairs([Y|XL]),
        create_pairs([X|XL]).
```

These `pair/2` constraints get checked for possible overlaps. By definition even one overlapping constraint can lead into a non-joinable final state of P. In the following figures all simplification rules for the `pair/2` constraints are shown.

Listing 3.12: Rule for pairs with propagation rules.

```
pair((_,_,==>,_,_),(_,_,==>,_,_)) <=>
        true.
```

Two propagation rules cannot result into a critical pair since both are applied when the constraints for their heads are given.

Listing 3.13: Rules for non-overlapping rules.

```
pair(([],RH1,_,_,_),(KH2,RH2,_,_,_)) <=>
        \+single_matching(RH1,KH2),
        \+single_matching(RH1,RH2) | true.

pair((KH1,[],_,_,_),(KH2,RH2,_,_,_)) <=>
        \+single_matching(KH1,KH2),
        \+single_matching(KH1,RH2) | true.

pair((KH1,RH1,_,_,_),(KH2,RH2,_,_,_)) <=>
        \+single_matching(KH1,KH2), \+single_matching(KH1,RH2),
        \+single_matching(RH1,KH2), \+single_matching(RH1,RH2) | true.
```

Pairs that do not overlap are removed from the constraint store. Rules are non-overlapping if all head constraints are different from each other. The three rules shown above are three different possible cases. In the first a simplification and another rule are checked. In the second a propagation and another. Finally in the third case a simpagation and another rule get checked for an overlap. In all cases no overlap is found.

Listing 3.14: Predicates for the matching of the head constraint of two different rules.

```
single_matching([H1|HL1],HL2) :-
        matching([H1],HL2,_).

single_matching([H1|HL1],HL2) :-
        not(matching([H1],HL2,_)),
        single_matching(HL1,HL2).
```

The structure of the `matching/3` predicate is shown in a later chapter for the `checker.pl`. The `single_matching/2` predicate succeeds if at least one head constraint of rule one is part of the head constraints of rule two. In that case one of the rules in the listings below succeeds and a `critical_pair/2` constraint is added to the constraint store.

Listing 3.15: Predicates for the matching of the head constraints of two different rules.

```
pair(([],RH1,RS1,G1,RHS1),(KH2,RH2,RS2,G2,RHS2)) <=>
        single_matching(RH1,KH2);
        single_matching(RH1,RH2) |
        critical_pair(([],RH1,RS1,G1,RHS1),(KH2,RH2,RS2,G2,RHS2)).

pair((KH1,[],RS1,G1,RHS1),(KH2,RH2,RS2,G2,RHS2)) <=>
        single_matching(KH1,KH2);
        single_matching(KH1,RH2) |
        critical_pair((KH1,[],RS1,G1,RHS1),(KH2,RH2,RS2,G2,RHS2)).

pair((KH1,RH1,RS1,G1,RHS1),(KH2,RH2,RS2,G2,RHS2)) <=>
        single_matching(KH1,KH2);
        single_matching(KH1,RH2);
        single_matching(RH1,KH2);
        single_matching(RH1,RH2) |
        critical_pair((KH1,RH1,RS1,G1,RHS1),(KH2,RH2,RS2,G2,RHS2)).
```

The guards of these rules got a disjunction between their predicates since even one matching head constraint of the two rules is enough for an overlap. After a critical pair has been found it gets printed out on the console.

The critical pairs that have been printed out on the console get removed from the CHR-constraint store and a `fin_critical_pair/2` with the same content is inserted into the store. In the next step all possible final critical pairs are removed from store and their content is inserted into the `cprl/2` constraint, a list where all overlaps are stored. The last action of `cp.pl` is calling the next program `checker.pl`.

3.3.3 Checking critical pairs: `checker.pl`

The `checker.pl` program is the core of the CC^p. All calculation before this point only prepared the input program to be check in this module. The input parameters of the `check/2` predicate are the overlaps of P and the list of rules of P.

The checker simulates an environment where CHR rules are fired like in a real SWI-Prolog calculation. It creates a list for the store where all constraints and built-ins are stored, a set of rules and the history constraints for the propagation history.

The check does not calculate the result for all permutations of the rule set – only for the permutation of the overlaps – the calculation can still detect confluence correctly since non-joinable states result from the application of an overlap. If the non-confluence is not fixed in this calculation we found a path to prove the non-confluence of P. If the program is confluent *all* paths will lead to the same final state.

This check also does not try different permutations for the variables. That is done while checking self-overlaps. If a rule would lead to a different result depending on the given order of the constraints this would be detected at that point and printed out on console as a self-overlap.

Listing 3.16: Predicate for initiating the confluence check.

```
1  check([],_) :- done.
2  check([(R1,R2)|RL],RuleSet) :-
3          copy_term(RuleSet,Rules), gensym(1,ID),
4          history([],ID,1), history([],ID,2),
5          build_store((R1,R2),ID),
6          make_first([R1],Rules,R1F), make_first([R2],Rules,R2F),
7          build_rules(R1F,ID,1), build_rules(R2F,ID,2),
8          conf(t,ID,(R1,R2)),
9          rule_list(Rules,ID), fin(ID),
10         check(RL,Rules).
```

To eliminate all possible side-effects of variable bindings between the calculations the list of rules is copied at the start of the computation. The gensym library is used the generate unique identifiers for rules, constraints and computations. Two `history/3` constraints are added to the CHR-constraint store – after this the store is built by using the head constraints of both rules of the overlap. This store is inserted into a `store/3` constraint. The `make_first/3` predicate ensured that one of the overlapping rules is the first that is fired in the computation. With these two orders of rules, `rule/8` constraints are built by the `build_rules/3` predicate. The `conf/3` constraint that is added to the CHR-constraint store after this is used to save the results of the joinability checks at the end of the computation. The `done/0` constraint in the base-case is used to print out the result of the confluence check after the calculation is done. The implementation of this is show in the output chapter.

Listing 3.17: Predicate for rule CHR-constraint generation.

```
build_rules([],_,_).
build_rules([(KH,RH,RS,G,RHS)|RL],ID,C) :-
        gensym(2,RID),
        rule(KH,RH,RS,G,RHS,RID,ID,C),
        build_rules(RL,ID,C).
```

The rules are generated from the rule-list in the first argument of the `build_rules/3` predicate. All rules get an unique identifier that is needed for the propagation history (`RID`) and an identifier for the current overlap (`ID`). The last argument of the `rule/8` constraint is the calculation flag. For each overlap two calculation are done. The rules need these flags since one of the overlapping rules is added to the constraint store first and thus firing first in each calculation.

Listing 3.18: Predicate for store CHR-constraint generation.

```
build_store([],_).
build_store(((KH1,RH1,_,G1,_),(KH2,RH2,_,G2,_)),ID) :-
        c_flat(KH1,K1F),
        c_flat(RH1,R1F),
        append(K1F,R1F,LHS1),
        c_flat(G1,G1F),
        append(LHS1,G1F,R1),
        c_flat(KH2,K2F),
        c_flat(RH2,R2F),
        append(K2F,R2F,LHS2),
        c_flat(G2,G2F),
        append(LHS2,G2F,R2),
        min_store(R1,R2,Store),
        build_id(Store,NStore),
        store(NStore,ID,1),
        store(NStore,ID,2).
```

To build the store the head constraints of both rules are needed. The minimal state that let both rules fire is built by the `min_store/3` predicate. All constraints that are used in the calculation get an unique identifier by the `build_id/2` predicate. Two identical stores are built, one for the calculation where the first rule is the first to be fired and one where the second is fired first.

When the program has created the rules and the store, the CHR-constraint store contains the all constraints that are needed to simulate a calculation in SWI-Prolog: `rule/8, history/3` and `store/3` constraints. These constraints have an unique identifier that guarantees that different calculation do not interact with each other.

The next step is the actual calculation. For this calculation all three types of CHR-rules need to be simulated – including all of the operations that are done by SWI-Prolog in the background.

These operations are logical steps like the check for matches between the head of a rule and the CHR-constraint store or even the propagation history.

If the program contains infinite calculations (like `a ⇔ a`) the CC^p will *not* terminate. There is no need to check the program for such rules since we demand a correct and terminating input program. When guards on multiple rules are used who cannot be understood by the CC^p it cannot give a clear result. Simple guards (like dimensions) are supported and evaluated. This is supported by checking the guards for complete different dimensions (like `X < 5` and `X > 5`). In this case the two rules can not lead to different final states with the same input constraints. For the cases where the guards are more complex the program is checked like there are no guards – if the final states of both calculations result into a critical pair the user is getting the message that the program is non-confluent if both guards can be applied at the same time.

To let the CC^p understand a bigger set of built-ins and let it really evaluate the built-ins – since it only works with patterns at the moment – a more complex support for built-ins needs to be added. Such a support was implemented by Frank Richter [10] in 2014. It was built on the Confluence Checker by Johannes Langbein [1]. Since the structure of the old checker and this new one is different in design and implementation the built-in support of Frank Richter cannot be used for this checker so easily. To use it the checker itself needs to be altered to support the structures of the built-in support. The already supported built-ins are mentioned in chapter 6.

The following rules show the solver for the CHR-rules of the calculation:

Listing 3.19: Rule for simplification rules.

```
1 rule([],RH,<=>,[],RHS,_,ID,C) \
2                  store(Store,ID,C), history(H,ID,C) <=>
3         matching(RH,Store,StoreMR) |
4         history(H,ID,C), build_id(RHS,NRHS),
5         append(NRHS,StoreMR,StoreRS),
6         apply_bindings(StoreRS),
7         find_equalities(StoreRS,StoreRS),
8         store(StoreRS,ID,C).
```

Listing 3.20: Rule for simpagation rules.

```
1 rule(KH,RH,<=>,[],RHS,_,ID,C) \
2                  store(Store,ID,C), history(H,ID,C) <=>
3         matching(KH,Store,_),
4         matching(RH,Store,StoreMR) |
5         history(H,ID,C), build_id(RHS,NRHS),
6         append(NRHS,StoreMR,StoreRS),
7         apply_bindings(StoreRS),
8         find_equalities(StoreRS,StoreRS),
9         store(StoreRS,ID,C).
```

If the rule has a guard a new CHR-constraint that contains the guard, the rule identifier and the calculation identifier is generated and the guard itself is added to the store. For simpagation rules the history is not build since the constraints in the removed head are different constraints every time the rule fires. The `apply_bindings/1` and the `find_equalities/2` predicates are looking for built-ins in the store that contain logical information (like `X=Y, X>=Y`) and apply those informations to the whole store. This check can only find simple logical information in the current implementation. While the `apply_bindings/1` predicate will apply all `X=Y` constraints the store the `find_equalities/2` predicate removes a pair of constraints that form an equivalence like `X>=Y, X=<Y` or `X>=Y, X=<Y` and insert a `X=Y` constraint to the store. This is done since a store with the information `(X>=Y, X=Y)` contains the *same* information like one with `(X=<Y, X=Y)`. Both lead to the same result: `X=Y`. The check for the equality of the stores at the end of the calculation will call the stores different if the equality checks are not done.

Listing 3.21: Rule for propagation rules.

```
1  rule(KH,[],==>,[],RHS,RID,ID,C) \
2                   store(Store,ID,C), history(H,ID,C) <=>
3          subset_match(KH,Store,NKH),
4          check_history(NKH,H,RID) |
5          build_history(NKH,H,RID,NHIS),
6          history(NHIS,ID,C),
7          build_id(RHS,NRHS),
8          append(NRHS,Store,StoreRS),
9          apply_bindings(StoreRS),
10         find_equalities(StoreRS,StoreRS),
11         store(StoreRS,ID,C).
```

The history contains all constraints that were applied to a given propagation rule. It is important that all of the head constraints are one element of the history list since by definition a propagation rule only fires once for the same combination of constraints in the same order.

If the calculation has stopped so that no rule can be applied any more, the `store/3` constraints are removed from the store and replaced by `result/3` CHR-constraints:

Listing 3.22: Rule for replacing the store constraints.

```
1  fin(ID) \ store(S,ID,C) <=>
2          msort(S,SSorted),
3          filter_store(SSorted,NewS),
4          result(NewS,ID,C).
```

The stores are sorted to make it easier to compare them and removed duplicate built-in constraints. The removing is done by the `filter_store/2` predicate. In this predicate the store is checked for multiple constraints that are equal. This is needed because the check for equiva-

lence of the states would result into different states even if the only differ in the number of `true` built-ins in the store. So only redundant information is removed.

Listing 3.23: Rule for checking the non-equivalence of states.

```
result(S1,ID,1), result(S2,ID,2), conf(_,ID,R) <=>
                 (not(matching(S1,S2,_));
                 not(matching(S2,S1,_))) |
                 conf(f,ID,R).
```

If the two states would be the same, the `matching/3` predicate would succeed in both ways. So the states are not equivalent if one of the matching predicates fail. In this case a `conf/3` CHR-constraint with the failing information is added to the store.

Listing 3.24: Rule for checking the equivalence of states.

```
result(S1,ID,1), result(S2,ID,2), conf(_,ID,R) <=>
                 matching(S1,S2,_),
                 matching(S2,S1,_) |
                 conf(c,ID,R).
```

If the two states are the same they match in both ways. In this case a `conf/3` CHR-constraint with the succeeding information is added to the store. This check ignores if the states have different variable names so `a(X)` and `a(Y)` are called the same.

Listing 3.25: Predicates for the matching.

```
matching([],R,R).

matching([C|CL],S,R) :-
        length(S,SL),
        unification(C,S,SL,RU),
        matching(CL,RU,R).
```

The matching predicate checks if the constraints of the first list (`[C|CL]`) are part of the second list (`S`) this predicate is widely used in the CC^p for every situation where it is important to know if one list is a subset of another the advantage of this predicate against built-ins like `subset/2` is, that no variable bindings are done and the predicate can find matches that are not found by `subset` since the structure of the constraints in CC^p is special in some cases and are only considered as identical because of the special use of this constraints in the context. The `subset` constraints can also not be used for the cases where the remaining list is needed after the elements of the first have been removed from the second. For this case the `subtract/3` predicate would come into mind. The problem with this predicate is the same like before, variable bindings are done. For the CC^p this would be fatal since the unique identifiers of the CHR-constraints would be bind to the constraints in the head of the rules. This is something that

is avoided by the `matching/3` predicate. With the `matching` predicate two lists can also be checked for equality:

Example 9. *Equality with* `matching`*: If two list should be checked for equality the* `matching/3` *predicate can be used like this:*

```
matching(List1,List2,_),
matching(List2,List1,_).
```

The only case where both predicates can succeed is if both list are equal to each other. Only thing that should be considered is that the matching predicate will call two constraints with different identifiers equal. This is needed for the check of the two states at the end of the calculation because constraints that only differ in their identifier are the logical the same.

When the `matching/3` predicate is used at the simulated rule application every constraint of the head is checked for occurrence in the store. The matching predicate fails if the constraints are not unifiable or the head constraints of the rule do not appear in the CHR-constraint store. In the last argument a list with all non matched constraints is given.

Listing 3.26: Predicates for the unification check.

```
1  unification(C,[S|SL],_,SL) :-
2          uni(C,S).
3
4  unification(C,[S|SL],L,R) :-
5          L >= 1, L1 is L-1, not(uni(C,S)),
6          append(SL,[S],NSL),
7          unification(C,NSL,L1,R).
8
9  uni(C,S) :- C == S.
10 uni(C,S) :-
11         compound(C), compound(S),
12         C =.. [A|CL], S =.. [A|SL],
13         length(CL,L), length(SL,L).
```

The `uni/2` predicate checks if the two constraints are unifiable without performing bindings. This is important cause we do not want that the head constraints of the rules get bound. This would result into a problem since the id of the constraints would be bound to the head constraints and other constraints of the store would not match any longer. Since the normal `uni/2` check does no cover a check for identical ids there has to be another check only for the history check. The length of the constraint store is used to know when all elements of the list have been checked. This is needed since we do not want to remove constraints from the store that should not. So the constraints that were already checked are appended to the end of the list.

Listing 3.27: Checking if the arguments of the constraints are unifiable.

```
1 tail_check([A|AL],[B|BL]) :-
2         (var(A); var(B)),
3         tail_check(AL,BL).
4
5 tail_check([A|AL],[B|BL]) :-
6         A == B,
7         tail_check(AL,BL).
```

This check is used in the `uni_subset/2` predicate which is only used for the history check.

3.4 Testcases and Examples

In this section some example input programs are given and their results when checked by the CC^p. Some examples are taken from [6] other are made to show special cases. The file name in the folder of the CC^p is given at each example. The overheads of the programs are cut out since they are trivial. These examples also have been used as testcases to check the functionality of the checker. All examples shown here are given in the folder of the CC^p.

Example 10. *simple.pl:*

```
1 p <=> q.
2 p <=> false.
3
4 Overlap: p <=> q with p <=> false
5 The states of p <=> q and p <=> false are non-joinable.
```

This basic example shows that the CC^p is able to find non-confluent overlaps. The store that is used in this example is `(p)` since this is the minimal state for both rules to fire.

Example 11. *coin.pl:*

```
1 throw(Coin) <=> Coin = head.
2 throw(Coin) <=> Coin = tail.
3
4 Overlap: throw(X) <=> X=head with throw(Y) <=> Y=tail
5 The states of throw(X) <=> X=head and throw(Y) <=>
6         Y=tail are non-joinable.
```

This example shows that the CC^p is able to work with built-ins. The variable bindings of the body of the rule are detected as different. The final states are `(Coin = head)` and `(Coin = tail)`.

Example 12. *self.pl:*

```
1 p(X), q(Y) <=> true.
2
3 The rule: p(X), q(Y) <=> true overlaps with itself and
4         results into a critical pair.
```

In this example the ability to detect self-overlaps is shown. The rule results into a critical pair since the overlap with the constraints `(p(X1), q(Y), p(X2))` would result into two different states: `(p(X1))` and `(p(X2))` depending on the order of the constraints. When looking at self-overlaps different variable names are detected and result into a non-joinable pair. This can be done since the variables are named the same at the beginning of the calculation.

Example 13. *max.pl:*

```
1 max(X,Y,Z) <=> X =< Y | Y = Z.
2 max(X,Y,Z) <=> X >= Y | X = Z.
3
4 Overlap: max(X,Y,Z) <=> X=<Y | Y=Z with max(A,B,C) <=> A>=B | A=C
5 The states of max(X,Y,Z) <=> X=<Y | Y=Z and max(X,Y,Z) <=>
6         X>=Y | X=Z are joinable.
```

In this example the support of guards is shown. When `X` and `Y` are equal both rules can fire, but since they have to be exactly the same value in this case the confluence property is not hurt. If `X` and `Y` are not the same only one of the rules could fire and the result is always be the same and not dependent on the order of the rules. The CC^p detects this by checking if both guards can be applied at the same time – in this case they can. The application of the guards is then applied on the constraint store, leading to `X = Y = Z`.

Example 14. *complex2.pl:*

```
1 p ==> q.
2 r, q <=> true.
3 r, p ,q <=> s.
4 s <=> p, q.
5
6 Overlap: r, q <=> true with r, p, q <=> s
7 Overlap: p ==> q with r, p, q <=> s
8 The states of r, q <=> true and r, p, q <=> s are non-joinable.
9 The states of p ==> q and r, p, q <=> s are non-joinable.
```

The example above shows that the CC^p supports propagation rules. It finds the overlap between the first and the third rule and detects the critical pair that results from this overlap. Since the

program terminates it can be seen that the propagation history stops the CC^p from applying a propagation rule multiple times with the same set of constraints.

Example 15. *prop2.pl:*

```
a(X), b(Y) <=> true.
a(X), b(Y) ==> c.

The rule: a(X), b(Y) <=> true overlaps with itself
        and results into a critical pair.

Overlap: a(X), b(Y) <=> true with a(X), b(Y) ==> c
The states of a(X), b(Y) <=> true and a(X), b(Y) ==> c
        are non-joinable.
```

This example shows how propagation rules are ignored when the program is checked for self-overlaps. Since propagation rules can fire once for each matching of the head constraints they will always result into the same final state.

Example 16. *simple1.pl*

```
a(X) <=> X < 5 | true.
a(X) <=> X > 5 | b(X).
a(X) <=> d(X).

Overlap: a(X) <=> X < 5 | true with a(X) <=> X > 5 | b(X)
Overlap: a(X) <=> X > 5 | b(X) with a(X) <=> d(X)
Overlap: a(X) <=> X < 5 | true with a(X) <=> d(X)
The states of a(X) <=> X < 5 | true and a(X) <=> X > 5 | b(X)
        are joinable.
The states of a(X) <=> X > 5 | b(X) and a(X) <=> d(X)
        are non-joinable.
The states of a(X) <=> X < 5 | true and a(X) <=> d(X)
        are non-joinable.
```

In this example the critical pair that results from rule three with the other rules is found while the other overlaps do not result into a critical pair thanks to the guards.

Example 17. *cell.pl*

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

The rule: assign(V,N), cell(V,O) <=> cell(V,N) overlaps with
        itself and results into a critical pair.
```

Example 18. *prop_conf.pl*

```
1 a, b ==> c.
2 a, a, b ==> c.
3 a, b, b ==> c.
4
5 No overlaps found - the program is confluent.
```

Theoretical three overlaps exist in the example – but since overlaps between propagation rules cannot result into non-joinable final states they are ignored.

4 The Propagation History

To implement the confluence check with all types of CHR-rules a history for already propagated CHR-constraints needs to be added. This chapter shows how the history was implemented into the CC^p.

4.1 Realising the History

When the CC^p has found an overlap and starts the confluence checking process inside `checker.pl`, two `history/3` CHR-constraints are added to the constraint store. These constraints contain an empty list at their first introduction, an unique identifier that is the same like the confluence checking process they are used in and a flag for being part of the first or the second calculation for this overlap. These constrains are updated every time a simulated CHR-rule application is done.

4.1.1 Building the History

For building the history the `build_history/4` predicate needs the constraints from the store that matched with the head of the rule. These constraints are found by the `subset_match/3` predicate. The first `build_history/4` predicate has the following structure:

Listing 4.1: Predicate for building the history when the set of constraints already fired.

```
1 build_history(X,H,RID,NH) :-
2         sort(X,XS), XH = (XS,AP,C,RID),
3         select(XH,H,XnH), APN is AP+1,
4         Y = (XS,APN,C,RID),
5         append([Y],XnH,NH).
```

The `X` contains the constraints that were used in the rule with the unique identifier `RID`. `H` is the already given propagation history. The constraints that should get added to the history are sorted since this is how they are stored inside the history. Since this is the case where this set of constraints already was applied to the rule and only a different order of constraints is used, there is a counter that gets increased by one. This counter is used to check if the set of constraint can still be applied to the rule.

Example 19. *Possibilities of propagations:*

If we got a propagation rule with multiple same constraints in the head like:

```
a, a, b, b ==> c.
```

then this propagation rule can be fired multiple times with the same set of constraints. In this case with the constraints `{a,a,b,b}`*. The maximum possibilities are calculated by the* `his_poss/2` *predicate. It calculates all possible permutations for the constraints. For the example above these permutations would be:*

$$\{a_1,a_2,b_1,b_2\},\{a_1,a_2,b_2,b_1\}\{a_2,a_1,b_1,b_2\}\{a_2,a_1,b_2,b_1\}$$

So the rule could fire four times with this set of four constraints.

Listing 4.2: Predicate for building the history when the constraints were not used before.

```
1 build_history(X,H,RID,NH) :-
2         sort(X,XS),
3         copy_term(XS,XSC),
4         unbind_id(XSC,XSU),
5         his_poss(XSU,C),
6         XH = (XS,_,C,RID),
7         not(select(XH,H,_)),
8         Y = (XS,1,C,RID),
9         append([Y],H,NH).
```

When the specific set of constraints was not be already applied to the rule with the unique identifier `RID` before a new history element is generated and the possible applications are calculated by the `his_poss/2` predicate. The counter for applications starts by one while the `C` for the maximum applications is at least one.

4.1.2 Checking the History

When a propagation rules can fire the CC^p will check the history if the rule already fired with this specific set of constraints. This check is part of the guard of the two rules for propagation rules. The check is only needed for propagation rules and not for simpagation rules since the used constraints of simpagation rules change every time they are applied because of the removed head constraints.

Listing 4.3: First predicate for checking the history.

```
1 check_history(_,[],_).
2 check_history(X,[H|HL],RID) :-
3         sort(X,XS),
4         H = (HH,_,_,RID2),
5         sort(HH,HS),
6         (XS \== HS; RID \== RID2),
7         check_history(X,HL,RID).
```

In this first predicate the constraints `X` from the store that matched the head of the rule with the unique identifier `RID` are not part of the history list. The constraints get sorted and the history element `H` is checked against this. If the constraints are not the same or the rule identifier is different the history element is different to the used constraints. In this case the next part of the history list is checked until the whole history got checked.

Listing 4.4: Second predicate for checking the history.

```
1 check_history(X,[H|HL],RID) :-
2         sort(X,XS),
3         H = (XS,AP,MAX,RID),
4         AP < MAX,
5         check_history(X,HL,RID).
```

If the combination of constraints and rule identifier is already part of the history there is still the possibility that it is a rule where the same constraints can be applied multiple times because of the structure of the rule (like in the example above). In this case this predicate finds an match between the constraints and the rule identifier. The element `H` of the history list contains the number of already done applications `AP` and the maximum possible applications `MAX`. As long as `AP` is smaller than `MAX` the rule can be applied. The counter `AP` is increased later by the `build_history/4` predicate.

5 Completion for CHR Programs

This section describes the implementation of the completion algorithm mentioned in [6]. For the implementation the simplification and deletion inference rules are not implemented since they already exist in the confluence check. The completion is started by the `completion/1` predicate. When started the CC^p performs a confluence check and calls the `completion/2` predicate in the `completion.pl` module if the program is not confluent. When the program reaches this state the simplification and deletion inference rule were already applied via the confluence check. This is guaranteed since the confluence check only stops when final states are reached and does not call the completion algorithm if the program is confluent. This also ensures that the completion terminates if a confluent and terminating set of rules has been found. The completion algorithm can also find no new rules because of the structure of the final states or run infinitely because of a new rule that let the whole program be non-terminating due to undecidability of halting problem. The rules for the completion algorithm can only find simple loops like rules that got the same constraints on the left and the right side.

5.1 Overview and Requirements

When trying to complete a program the the bold typed modules and files are now used additionally to the ones of the confluence check.

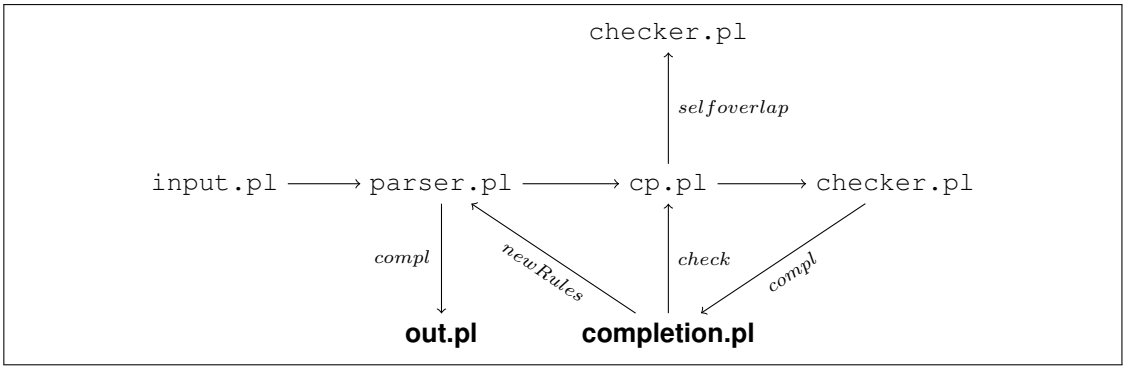

Figure 5.1: The program flow of the CC^p and its output.

In the figure one possible loop is already shown: A program that is checked for confluence and is completed but never reaches a set of rules that is confluent.

To implement the completion algorithm the following requirements must be met by the CC^p:

- read the critical-pairs *CP* and the rule list R of P as input.
- apply the rules of the completion algorithm (Simplification, Deletion, Orientation, Introduction).
- understand when to use which rule of the algorithm.

Like the confluence checker the completion algorithm needs a correct and terminating CHR-program to terminate. While completing the algorithm itself can produce rules that generate infinite loops.

5.2 Changes to the program files of the confluence check

Let P be a syntactically correct and terminating non-confluent CHR-program then the CC^p can try to complete P to make it confluent. For implementing the completion algorithm all parts of the CC^p got minor changes.

5.2.1 Changes to `parser.pl`

To call the completion algorithm `completion([File Path])` needs to be called. This works like the `confluence/1` predicate but adds a `compl/0` CHR-constraint to the store. This constraint is needed to fire the rule that creates a file where the completed program is saved. The parser also has rules and predicates to ensure the writing of the new rules into the `out.pl` file. These rules are shown in the output chapter.

5.2.2 Changes to `cp.pl`

In `cp.pl` a `compl/0` CHR-constraint is added to the store to indicate that the completion has been called. In `cp.pl` this constraint is used to stop the program from writing out that the program is confluent when no overlap is found and it lets `cp.pl` call a different predicate to start the `checker.pl`.

In this module the rules are also checked for duplicates. This is done by the `filter_rules/2` predicate. Since the different overlaps can have the same non-joinable final states it is possible that the same rule gets generated multiple times. This happens because there is no detection for this inside the `checker.pl` since checks can be done parallel and calculation can be in different states at the same time. The other parts of the `cp.pl` program are working like in the confluence check.

5.2.3 Changes to checker.pl

In checker.pl the compl/0 constraint is also added. In the end of the computation the following rule fires if the program is non-confluent:

Listing 5.1: Rule to find non-confluent programs while calling the completion algorithm.

```
compl \ result(S1,ID,1), result(S2,ID,2), conf(_,ID,R) <=>
                filter_store(S1,NS1),
                filter_store(S2,NS2),
                (not(matching(NS1,NS2,_));
                not(matching(NS2,NS1,_))) |
                cp(NS1,NS2),
                conf(f,ID,R).
```

The cp/2 CHR-constraint stores the non-joinable final states of the computation. When this constraint is generated another rule is fired that calls the completion algorithm:

Listing 5.2: Rule to start the completion algorithm (output parts removed).

```
rule_list(RL,ID), compl \ conf(f,ID,((KH1,RH1,RS1,G1,RHS1),
        (KH2,RH2,RS2,G2,RHS2))), cp(S1,S2) <=>
                [...]
                completion(RL,(S1,S2)),
                cleanup(ID).
```

5.3 Completion a program: completion.pl

In completion.pl the algorithm that is described in [6] is implemented. This part of the program is written in Prolog.

Listing 5.3: Main predicate of the completion algorithm.

```
completion(RL,(S1,S2)) :-
        orientation((S1,S2),([],RHS,RSS,GS,RHSS),(KHP,[],RSP,GP,RHSP)),
        introduction(([],RHS,RSS,GS,RHSS),(KHP,[],RSP,GP,RHSP),RL).
```

Like mentioned at the start of the chapter the simplification and the deletion inference rule are not implemented in the completion part since they are already implemented in the confluence check. The orientation/3 predicate creates none, one or two rules, based on the non-joinable final state pair. The introduction predicate adds the new rules to the rule list, adds them to the out.pl file and calls another confluence check on the new set of rules.

Listing 5.4: Main predicate for orientation.

```
1 orientation((S1,S2),NRS,NRP) :-
2         remove_true(S1,S1wT),
3         remove_true(S2,S2wT),
4         create_simp((S1wT,S2wT),NRS),
5         create_prop((S1wT,S2wT),NRP).
```

The `orientation/3` predicate tries to create a simplification and a propagation rule. Before this is done the `true/0` constraints have to be removed from the store – they do not contain any information and would make the predicates to create new rules more complex or even generate errors. When which rule can be created is described in [6, p. 113] and chapter 2.8.1, it is depending on the content of the store.

Listing 5.5: Predicates for creating a no new simplification rule.

```
1  create_simp((S1,S2),NRS) :-
2          filter_cons(S1,S1C),
3          filter_cons(S2,S2C), S1C = S2C,
4          NRS = ([],[],[],[],[]).
5  create_simp((S1,S2),NRS) :-
6          filter_cons(S1,S1C),
7          filter_cons(S2,S2C),
8          matching(S1C,S2C,_),
9          matching(S2C,S1C,_),
10         NRS = ([],[],[],[],[]).
11 create_simp((S1,S2),NRS) :-
12         filter_buil(S1,S1B),
13         filter_buil(S2,S2B),
14         eq_check(S1B,S2B),
15         NRS = ([],[],[],[],[]).
```

The first predicate is for the case where the CHR-constraints in the store are the same, a new simplification rule with those informations would create an infinite loop. In the second case both stores are not exactly the same but would still loop like in: `a,b` $\Leftrightarrow$ `a,b,c`. The new `a` and `b` CHR-constraints will cause the rule to fire again and again. To detect this `matching/3` is used. This case is solved in later predicates by switching the left and the right side of the rule. In the third case the built-ins of both states lead to a contradiction. Since one of the built-ins would be in the guard and the other in the body they would lead to a non-consistent store. All three cases would lead to rules we could not use or would make the computation non-terminating. In these cases the completion algorithm cannot find a new rule using the given final states. If the three cases above do not apply a simplification rule is created using the following predicate:

Listing 5.6: Predicate for creating a new simplification rule.

```
create_simp((S1,S2),NRS) :-
        filter_cons(S1,S1C), filter_cons(S2,S2C),
        filter_buil(S1,S1B), filter_buil(S2,S2B),
        unbind_id(S2C,S2U),
        unbind_id(S1C,S1U),
        S1U \= [],
        append(S2U,S2B,S2Co),
        empty_check(S2Co,S2NE),
        not(matching(S1U,S2NE,_)),
        NRS = ([],S1U,<=>,S1B,S2NE).
```

A new rule is created by splitting the store into built-ins and CHR-constraints. An empty check is done for the body of the rules if the body is empty a `true/0` predicate is inserted into the body by the `empty_check/2` predicate. There is also a second predicate in which `S1` and `S2` switch places. The switching is done if a rule like mentioned before would be created (`a,b` ⇔ `a,b,c`) the right and the left side get switched to form the rule `a,b,c` ⇔ `a,b`. This rule does not create an infinite loop and this method is mentioned as valid in [6].

While creating propagation rules we have to check if redundant information is created or the built-ins lead to a contradiction. A new propagation rule should give us new information which leads to a joinable pair of states.

Listing 5.7: Rules for creating no propagation rule.

```
create_prop((S1,S2),([],[],[],[],[])) :-
        ((filter_buil(S1,[]),filter_buil(S2,[]));
        (filter_buil(S1,S1B),filter_buil(S2,S2B),
        eq_check(S1B,S2B))).

create_prop((_,S2),([],[],[],[],[])) :-
        filter_buil(S2,S2B),
        not(false_check(S2B)).

create_prop((_,S2),([],[],[],[],[])) :-
        filter_cons(S2,[]).
```

These are the cases where no propagation rule is created. In the first case there are either no built-ins in the store or the built-ins contain the same information. A new propagation rule would create no new information or none at all. In the second case a guard that contains the predicate `false/0` would be created. This rule would be never able to fire. In the last case the head of the rule would be empty – this is impossible for CHR-Rules by definition.

Listing 5.8: First rule for creating a new propagation rule.

```
create_prop((S1,S2),NRS) :-
        filter_cons(S2,S2C),
        filter_buil(S1,[]), filter_buil(S2,S2B),
        S2B \= [],
        false_check(S2B),
        unbind_id(S2C,S2U),
        S2U \= [],
        NRS = (S2U,[],==>,S2B,[true]).
```

In this case a new rule is created where built-ins for a guard are in the store but none for the right hand side of the rule. Because of the structure of rules representation in the CC^p we need to add a `true` to the right hand side. A false check for the guard is done since a rule with a false guard would never be able to fire. The `false_check/1` simply goes through all built-ins in the list `S2B` and checks if the are `false`.

Listing 5.9: Second rule for creating a new propagation rule.

```
create_prop((S1,S2),NRS) :-
        filter_cons(S2,S2C),
        filter_buil(S1,S1B),
        S1B \= [],
        filter_buil(S2,[]),
        unbind_id(S2C,S2U),
        S2U \= [],
        NRS = (S2U,[],==>,[],S1B).
```

In this case the new rule does not have a guard since there are no built-ins in the store to create one. So this rule will only create new information which is added to the constraint store. Like in all predicates that try to build a new propagation rule this predicate checks if the head of the rule (`S2U`) or the right hand side (`S1B`) would be empty. In this case this predicate is not applied.

Listing 5.10: Third rule for creating a new propagation rule.

```
create_prop((S1,S2),NRS) :-
        filter_cons(S2,S2C),
        filter_buil(S1,S1B), filter_buil(S2,S2B),
        not(eq_check(S1B,S2B)),
        S1B \= [], S2B \= [],
        false_check(S2B),
        unbind_id(S2C,S2U),
        S2U \= [],
        NRS = (S2U,[],==>,S2B,S1B).
```

This is the case where information for a guard and and a right hand side is existing. A new propagation rule with all of the informations is created. Sometimes this predicate can create a rule where the guard and the right hand side are identical. Since this is no contradiction and redundant information gets filtered at the confluence check this cases can be ignored and does not need to be filtered here.

The used `filter_cons/2` and `filter_buil/2` predicates look up the CHR-constraints in their definition that is typically at the beginning of the program. The information was generated at the `parser.pl` and is added as a CHR-constraint. They use the `find_chr_constraint/1` predicate to find this information inside the constraint store.

Listing 5.11: Rules for the introduction of new rules.

```
1  introduction(([],[],[],[],[]),([],[],[],[],[]),_) :-
2          writeln('No new rules could be created!').
3
4  introduction(NRS,([],[],[],[],[]),RL) :-
5          append([NRS],RL,Rules),
6          transform_rule(NRS,NNR1),
7          compl_file([NNR1]),
8          critical_pairs_com(Rules,Rules).
9
10 introduction(NRS,NRP,RL) :-
11         append([NRS],RL,R1),
12         append([NRP],R1,Rules),
13         transform_rule(NRS,NNR1),
14         transform_rule(NRP,NNR2),
15         compl_file([NNR1,NNR2]),
16         critical_pairs_com(Rules,Rules).
```

In the introduction inference rule three cases are possible, no rule, one rule or two rules. For no rule trivially no new rule is introduced. The user is getting feedback on console so that he knows what happened and why no new rule appears on the screen. The second case is where only a simplification rule is generated – in this case the rules is inserted into the rule list and written into the `out.pl` file where the the completed program is saved. In the second case two rules were created and are added to the rule list and the output program.

The `transform_rule/2` predicate is a set of facts who transform the rule presentation of this part of the program into the presentation that is needed by the predicate which writes the rules into the `out.pl` file.

5.4 Examples and Testcases

In this section the non-confluent examples from the confluence checker chapter are used as well as examples from [6]. All examples that are shown here are given in the folder of the CC^p. If variables were part of the constraint their names have been replaced since SWI-Prolog is using names like `_G123` by default.

Example 20. *simple.pl:*

```
p <=> q.
p <=> false.

Overlap: p <=> q with p <=> false
The states of p <=> q and p <=> false are non-joinable.
Trying to complete ...
Trying with the new rule(s):
q <=> false

Overlap: p <=> q with p <=> false
The states of p <=> q and p <=> false are joinable.
```

After the confluence check the critical pair `(q;false)` is found. This pair results directly into the new rule. With the predicates for the creation of new rules the pair `(false;q)` would also find this rule.

Example 21. *coin.pl:*

```
throw(Coin) <=> Coin = head.
throw(Coin) <=> Coin = tail.

Overlap: throw(Coin) <=> Coin=head with throw(Coin) <=> Coin=tail
The states of throw(Coin) <=> Coin=head and throw(Coin) <=>
        Coin=tail are non-joinable.
Trying to complete ...
Trying with the new rule(s):

No new rules could be created!
```

In this example no new rules can be created since the critical pair is `(Coin=head;Coin=tail)`. Since in both states only built-ins exist we cannot build a CHR-rule since it would need a constraint to be able to fire – the critical pair cannot be fixed by the completion algorithm.

Example 22. *complex2.pl:*

```
1  p ==> q.
2  r, q <=> true.
3  r, p, q <=> s.
4  s <=> p ,q.
5
6  The states of r, q <=> true and r, p, q <=> s are non-joinable.
7  Trying to complete ...
8  Trying with the new rule(s):
9  p, q, q <=> p, q
10
11 The states of r, p, q <=> s and p, q, q <=> p, q are non-joinable.
12 Trying to complete ...
13 Trying with the new rule(s):
14 p, q <=> p
```

This example program is a case where the completion algorithm creates an infinite loop that cannot be detected by simple matchings. In the computation the algorithm will create the new rule `p,q,q ⇔ p,q`. This new rule leads to new critical pairs where the rule `p,q ⇔ p` is newly created. When the confluence checker is checking the first rule and this rule for confluence it will never stop because the new rule produces a `p/0` constraint which can be propagated and a `q/0` constraint is created. Now the `p/0` constraint and the `q/0` constraint can be applied to the new rule and so on. The problem with this is that the CC^p cannot detect this problem in its current implementation and will calculate until SWI-Prolog is out of global stack.

It is possible so write a program that stops this behaviour but due to the complexity of this problem it is no subject of this work. A program that checks the connecting between all rules would be needed that can detect cycles and thus stop the completion algorithm from creating that specific rule.

Example 23. *self.pl:*

```
1  p(X), q(Y) <=> true.
2
3  The rule: p(X), q(Y) <=> true overlaps with itself and results
4          into a critical pair.
```

Since the critical pair of a self overlapping rule is something like `(p(X);p(Y))` we cannot build a proper new rule that would fix this critical pair. A program with only one rule never gets checked in the completion algorithm since the generated rule would result in an infinite loop.

Example 24. *prop2.pl:*

```
a(X), b(Y) <=> true.
a(X), b(Y) ==> c.

The rule: a(X), b(Y) <=> true overlaps with itself and results
        into a critical pair.

Overlap: a(X), b(Y) <=> true with a(X), b(Y) ==> c
The states of a(X), b(Y) <=> true and a(X), b(Y) ==> c are
        non-joinable.
Trying to complete ...
Trying with the new rule(s):
c <=> true

The rule: a(X), b(Y) <=> true overlaps with itself and results
        into a critical pair.

Overlap: a(X), b(Y) <=> true with a(X), b(Y) ==> c
The states of a(X), b(Y) <=> true and a(X), b(Y) ==> c are joinable.
```

Like mentioned before the completion algorithm will not fix the self overlap which results into a critical pair. But it will fix the critical pairs that resulted from the non-joinable states of the two rules even when a self-overlap was already found.

Example 25. *simple1.pl:*

```
a(X) <=> X < 5 | true.
a(X) <=> X > 5 | b(X).
a(X) <=> d(X).

Overlap: a(X) <=> X<5 | true with a(X) <=> X>5 | b(X)
Overlap: a(X) <=> X>5 | b(X) with a(X) <=> d(X)
Overlap: a(X) <=> X<5 | true with a(X) <=> d(X)
The states of a(X) <=> X>5 | b(X) and a(X) <=> d(X) are non-joinable.

Trying to complete ...
Trying with the new rule(s):
b(X) <=> X>5|d(X), X>5
d(X) ==> X>5|X>5
```

```
1  Trying to complete ...
2  Trying with the new rule(s):
3  b(X) <=> X>5|d(X), X>5
4  d(X) ==> X>5|X>5
5
6  The states of a(X) <=> X<5 | true and a(X) <=> d(X) are non-joinable.
7  Trying to complete ...
8  Trying with the new rule(s):
9  d(X) <=> X<5|X<5
10 d(X) ==> X<5|X<5
11 Overlap: d(X) ==> X<5 | X<5 with d(X) <=> X<5 | X<5
12 Overlap: a(X) <=> X<5 | true with a(X) <=> X>5 | b(X)
13 Overlap: a(X) <=> X>5 | b(X) with a(X) <=> d(X)
14 Overlap: a(X) <=> X<5 | true with a(X) <=> d(X)
15 Overlap: a(X) <=> X>5 | b(X) with a(X) <=> d(X)
16 Overlap: a(X) <=> X<5 | true with a(X) <=> X>5 | b(X)
17 Overlap: a(X) <=> X>5 | b(X) with a(X) <=> d(X)
18 Overlap: a(X) <=> X<5 | true with a(X) <=> d(X)
19 Overlap: a(X) <=> X>5 | b(X) with a(X) <=> d(X)
20 The states of d(X) ==> X<5 | X<5 and d(X) <=> X<5 | X<5 are joinable.
21 The states of a(X) <=> X<5 | true and a(X) <=> X>5 | b(X) are joinable.
22 The states of a(X) <=> X>5 | b(X) and a(X) <=> d(X) are joinable.
23 The states of a(X) <=> X<5 | true and a(X) <=> d(X) are joinable.
24 The states of a(X) <=> X>5 | b(X) and a(X) <=> d(X) are joinable.
25 The states of a(X) <=> X<5 | true and a(X) <=> X>5 | b(X) are joinable.
26 The states of a(X) <=> X>5 | b(X) and a(X) <=> d(X) are joinable.
27 The states of a(X) <=> X<5 | true and a(X) <=> d(X) are joinable.
28 The states of a(X) <=> X>5 | b(X) and a(X) <=> d(X) are joinable.
```

This example shows that the CC^p has a understanding of guard logic. In the first part the confluence check notices that the first and the second rule cannot result into a critical pair because their guards eliminate the possibility that both rules can fire with the same store. But both rules overlap with the third cause it has no guard. In the overlap between rule two and three two new rules are added to the program but they both only fire if the logical information of the guard of rule two is satisfied. With this rules the information `X>5` is added to the constraint store even when rule three fires instead of rule two. These two rules fix the critical pair `((d(X)),(X>5,b(X)))` since the propagation rule will add `X>5` to the store and the simplification rule will remove `b(X)` and add `d(X)`.

To fix the second critical pair two rules are added who only fire when the guard of rule one is satisfied. With these rules the critical pair `((X<5),(d(X)))` will get fixed since the second state will be changed to `X<5` by the simplification rule. In this case the propagation rule is not needed at both calculation but generated because of the inference rules of the completion algorithm.

6 Support for simple built-ins for the confluence checker

This chapter describes how and which simple built-ins are supported in the CC^p. The built-ins that are supported currently are: `<,>,=<,>=,=,\=`. More could be added to the computation by implementing a complex guard check and variable management sub-program. However the focus of this work is to implement a fully functional confluence checker and the completion algorithm.

The support for built-ins is only implemented in `checker.pl` and `completion.pl` since these are the only parts of the program where the logic of the built-ins can be used.

6.1 Bindings using: `=, =<, >=`

After each application of a rule the `apply_bindings/1` predicate is called. The predicate searches for equivalence constraints in the store like `X = Y`.

Listing 6.1: Predicates for the application of equivalences.

```
1 apply_bindings([]).
2
3 apply_bindings([(X = Y)|RL]) :-
4         X = Y,
5         apply_bindings(RL).
6
7 apply_bindings([_|RL]) :-
8         apply_bindings(RL).
```

The predicate will apply every binding that is mentioned in the store. The binding will also be applied to every constraint in the store that contains the variables. Another part where bindings are done is when the binding is a logical consequence of two constraints like `X >= Y, X =< Y`. If both predicates are applicable at the same time `X = Y` has to be satisfied. These constraints do not get removed from the store in the process.

Listing 6.2: Predicates for finding equivalences.

```
find_equalities([],_).
find_equalities([_],_).
find_equalities([X =< Y|RL],S) :-
        member((X >= Y),RL),
        X = Y,
        update_store(X = Y,S),
        find_equalities(RL,S).

find_equalities([X >= Y|RL],S) :-
        member((X =< Y),RL),
        X = Y,
        update_store(X = Y,S),
        find_equalities(RL,S).

find_equalities([_|RL],S) :-
        find_equalities(RL,S).

update_store(X = Y, _) :-
        X = Y.
```

If there are multiple equivalent binding expressions in the store one of them is removed. This is needed since the state equivalence check at the end of the computation would call states like `(X=Y,X=Y;X=Y)` different. Since the multiple occurrence of the same logical expression is redundant information the store contains the exact same information after removing these duplicates.

6.2 Guarded confluence: `>, <, =, \=`

A property called *guarded confluence* in this context is a case where the confluence of a program is ensured by the guards of the rules. Let R_1 and R_2 be two rules that overlap with each other. If these rules both contain a guard and these guards can never be satisfied at the same time then these rules ensure guarded local confluence.

Examples for these guards are the pairs: `(X > Y, X < Y)`, `(X = Y, X \= Y)`. In these cases only one of the guards can be satisfied at the same time. The confluence check for this overlap will be cancelled since there is no possible overlap where both rules could fire. The finding of these guards in realised by a list of facts with all possible cases. The declaration of local confluence for these overlaps is realised by one rule:

Listing 6.3: Rule for guarded confluence.

```
result(_,ID,1), result(_,ID,2), guard([G1],RID1,ID,1),
        guard([G2],RID2,ID,2) <=>
                check_equation(G1,G2) | guarded_conf((RID1,RID2),ID).
```

The results of the confluence check are not interesting since the guards guarantee local confluence for the given overlap.

6.3 Completing with satisfiable guards

In the `completion.pl` part of the CC^p new rules are generated. Since these new rules contain the built-ins from both states of the critical pairs these built-ins can lead to a contradiction. To prevent this a `eq_check/2` predicate is used to check if the built-ins lead to such a contradiction:

Listing 6.4: Predicates and facts for checking two relations for a contradiction.

```
eq_check([],_) :- false.
eq_check(_,[]) :- false.
eq_check([X|_],[Y|_]) :-
        check_equation(X,Y).
eq_check([X|XL],[Y|YL]) :-
        not(check_equation(X,Y)),
        (eq_check(XL,[Y|YL]);
        eq_check([X|XL],YL)).

check_equation(X < Y, X > Y).
check_equation(X > Y, X < Y).
check_equation(X >= Y, X < Y).
check_equation(X =< Y, X > Y).
check_equation(X < Y, X >= Y).
check_equation(X > Y, X =< Y).
check_equation(X = Y, X < Y).
check_equation(X = Y, X > Y).
check_equation(X > Y, X = Y).
check_equation(X < Y, X = Y).
```

The `eq_check/2` predicate is true if a contradiction is found, in this case no new rule can be created with the given built-ins of the final non-joinable state.

7 Output

Like mentioned in a chapter before the CC^p is working in a pipeline (figure below). In this chapter all outputs in the individual modules are shown. Additional the code generation for the output file (`out.pl`) is shown. The bold typed modules are the ones that get discussed in this chapter. The `parser.pl` module itself is not marked but it creates the `out.pl` file.

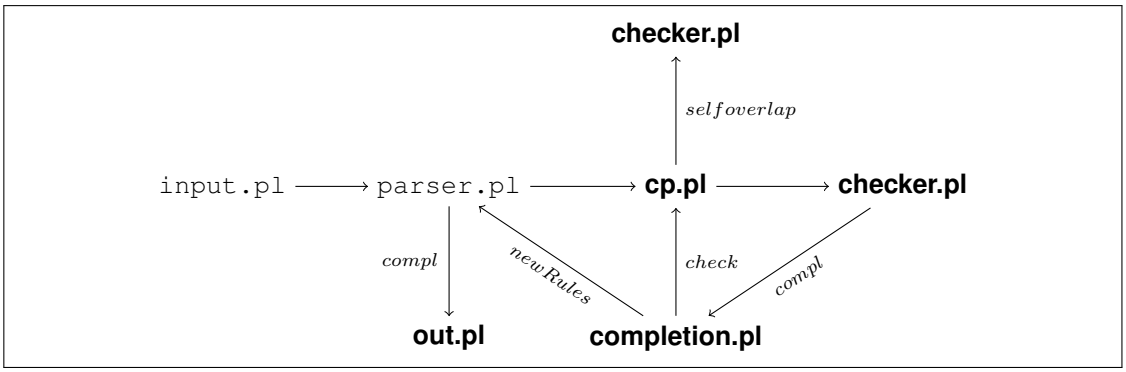

Figure 7.1: The program flow of the CC^p and its output.

7.1 Output while checking confluence

When `confluence([File Path])` is called the CC^p wont go to the step that is marked with *compl*. It will check the program for (self-)overlaps in `cp.pl`, if (self-)overlaps are found the following output will be shown on the console before starting the check for confluence:

Listing 7.1: Possible outputs given by `cp.pl` when overlaps are found.

```
1 The rule: [R] overlaps with itself and results into a critical pair.
2
3 Overlap [R1] with [R2].
```

The advantage of showing the overlaps before starting the confluence check is that the user can detect if a specific overlaps end in an infinite loop. It will just show the overlap and calculates infinite. If no (self-)overlaps are found the following output will be printed out on console:

Listing 7.2: Output given by `cp.pl` when no (self-)overlap is found.

```
1 No overlaps found - the program is confluent.
```

The `checker.pl` will only print out lines when it has a result for a computation. The first one only occurs if the guards of two rules can be applied at the same time and the states are non-joinable. This output is only shown when the guards are not part of the supported built-ins mentioned in chapter 6.

Listing 7.3: Possible outputs given by `checker.pl`.

```
1 The states result into a critical pair if [G1] and [G2]
2         can be applied at the same time.
3
4 The states of [R1] and [R2] are non-joinable.
5
6 The states of [R1] and [R2] are joinable.
```

If the calculation does not stop because of an infinite loop no output of `checker.pl` is shown since it needs a finished calculation to compare the results. This case can happen if the input program contained such a rule or the completion algorithm created one.

After the calculation is done all local results from all overlaps are collected. A confluent pair generates a `c/1` constraint that contains the confluence information. The non-confluent pairs generate `c/1` constraints that contain the non-confluence information. If two `c/1` constraints contain the same information one is removed from the store. If one of each type is inside the constraint store the one with the positive confluence information is removed from the store. When only one `c` constraint is left the result is printed out on console:

Listing 7.4: Possible outputs for the confluence property of a program.

```
1 The program IS confluent.
2
3 The program is NOT confluent.
```

For completion this output is disabled.

7.2 Output while completing

When the calculation of the CC^p is started by `completion([File path])` the calculation will do all of the steps in figure 7.1 if the program is non-confluent. The output of the parts that are used for the confluence check does not change. The `completion.pl` part itself prints out the following lines:

Listing 7.5: Possible outputs given by `completion.pl`.

```
Trying to complete ...
Trying with the new rule(s):

[Simplification Rule]
[Propagation Rule]

No new rules could be created!
```

The first output is shown every time the computation is entering the completion step. The new rules are only shown if they are really created. If no new rule is created the last message is printed out.

A problem with the completing of larger program is that the user cannot copy all the rules that are printed out on the console. To address this problem the `out.pl` file is created every time a completion is started. The file contains the original program and the new rules as well. For this case all the lines that were read in `parser.pl` are stored in the CHR-constraint store. There are three type of lines that are stored `module/1, chr_constraints/1` and `rule/1` constraints. If a new rule is created the `introduction/3` predicate calls the `compl_file/1` predicate in `parser.pl`. It creates a new `rule/1` constraint and a new `stream/2` constraint.

Listing 7.6: Predicates for adding new rules to `out.pl`.

```
compl_file([]) :- stream(_,write), compl.
compl_file([NR|NRL]) :-
        find_chr_constraint(c_rule(OldR)),
        same_checker(NR,OldR), compl_file(NRL).
compl_file([NR|NRL]) :-
        find_chr_constraint(c_rule(OldR)),
        not(same_checker(NR,OR)),
        rule(NR), c_rule(NR), compl_file(NRL).
```

The new rule is only added if it is not already a part of `out.pl`, how this is done will be described later.

For the different types of constraints that need to be written into the file and for the different types of rules unique rules are given:

Listing 7.7: Rule for writing the module declaration into the file.

```
compl \ stream(S,write), module(M) <=>
        open('out.pl',append,S),
        write(S,':- use_module('), write(S,M),
        write(S,').\n'), close(S),
        stream(_,write).
```

Even if the CC^p does not support many built-ins in its currents implementation the ability to parse the used modules is done for possible future extensions.

Listing 7.8: Rule for writing the CHR-constraint definition into the file.

```
compl \ stream(S,write), chr_constraints(C) <=>
        open('out.pl',append,S),
        write(S,':- chr_constraint '), l_write(S,C),
        write(S,'.\n'), close(S),
        stream(_,write).
```

When the rules are written into the `out.pl` file the ids are removed from the rules. This is done by the `remove_id/2` predicates.

Listing 7.9: Rule for writing a simpagation rule with a guard into the file.

```
compl \ stream(S,write), rule((KH \ RH <=> G | RHS)) <=>
        remove_id(KH,RKH), remove_id(RH,RRH), remove_id(RHS,RRHS),
        open('out.pl',append,S),
        l_write(S,RKH), write(S,'\\'),
        l_write(S,RRH), write(S,<=>), write(S,'\n'),
        l_write(S,G), write(S,'|'),
        l_write(S,RRHS), write(S,'.\n'), close(S),
        stream(_,write).
```

The `compl/0` CHR-constraint is used as an indicator that the completion has been started. It is created by the `completion/1` predicate. There are six rules for writing the CHR-rules into the file, two for every rule type one with a guard and one with none.

A problem that is resulting from the parallelism of the computation is that it is possible that a rule is created more than one time by the completion algorithm. This happens if more than one overlap leads into the same final critical state pair. If there is no check for duplicates they would be added to `out.pl`. A program where every overlap generates the same final critical state pair would contain many instances of the same rule – since this would distract the user a check for duplicates is done before the new rules get added to `out.pl`. This is done by creating a `c_rule/1` constraint is created for every `rule/1` constraint.

When the `compl_file/1` predicate is called a `find_chr_constraint/1` predicate searches for a `c_rule/1` constraint in the CHR-constraint store where the `same_checker/2` predicate is true. When no `c_rule/1` that fulfils the predicate is found the rule is not a part of `out.pl` and will be added to it.

When the rules get written into the file the variables do not have an explicit name. Their names will be choose by SWI-Prolog like in the following example:

Example 26. *simple.pl:*

```
Input program:

p(X) <=> q.
p(X) <=> false.

Completed program:

p(_G12069)<=>q.
p(_G12360)<=>false.
q<=>false.
```

8 Evaluation

In the following section the performance and the correctness of the CC^p will be checked. The runtime and the results of different examples will be compared to the runtime and results of the confluence checker written by Johannes Langbein[1]. The performance check was done to show how long the calculation can last when programs are more complex.

To check the runtime the `time/1` Prolog predicate is used. Since a program can have different runtimes on each execution the examples will be tested twenty times. The shown times are the arithmetic mean. If there is an execution that needed noticeable more time (like one second more than the others) that the others it was replaced by a additional execution. While testing the examples the checker [1] had a significant increase of runtime after several executions in the same SWI-Prolog instance. The runtimes were on the same level like the first executions after starting a fresh instance of SWI-Prolog.

If an examples needed had a long calculation time it was not cancelled until SWI-Prolog ran out of global stack. An exception to this are the completion examples where the completion algorithm creates an infinite loop.

Every example program that is used in this chapter is part of the content that is delivered together with the CC^p. The examples are divided into confluent, non-confluent and increasing complexity examples.

8.1 Machine

For the tests all unnecessary processes on the machine got terminated.

Components	**Machine**
CPU	Intel i7-4770 @ 3.4 GHz
RAM	8.00GB DDR3 (1600 MHz)
OS	Windows 10 (x64)
Storage	SSD

The SWI-Prolog process was launched under normal priority. The used version was 7.1.29, 64-bit edition.

8.2 Tests (Confluence)

The following tests were done by calling the predicate `time(confluence('File Path'))` for the CC^p and `time(check_confluence('File Path'))` for the Checker by Johannes Langbein. If the test program contains a propagation rule no runtime for the checker can be evaluated since it does not support this type of rules.

Confluent Ex.	Rules	CC^p	Checker [1]	CC^p - Checker	Overl.	Same Result
min.pl	2	0.0155s	0.0276s	- 0.0121s	1	Yes
prop_conf.pl	3	0.0081s	–	–	0	–[1]
simple.pl[3]	2	0.0103s	–	–	1	–[1]
simple2.pl	2	0.0177s	0.0386s	- 0.0209s	1	Yes
simple3.pl	2	0.0104s	–	–	0	–[1]
simple4.pl	2	0.0011s	0.0280s	- 0.0269s	0	Yes
simple5.pl[3]	3	0.0341s	0.0229s	+ 0.0112s	3	Yes
simple6.pl[3]	2	0.0017s	0.0019s	- 0.0002s	Yes	
complex.pl	7	0.0411s	0.1108s	- 0.0697s	11	Yes
complex2.pl	12	0.1571s	0.47685s	- 0.3198s	9	Yes
max.pl	2	0.0078s	0.0215s	- 0.0137s	1	Yes

N.-Confl. Ex.	Rules	CC^p	Checker [1]	CC^p - Checker	Overl.	Critical	S. Res.
cell.pl[3]	1	0.0185s	0.2332s	- 0.2147s	1	1	Yes
coin.pl[3]	2	0.0134s	0.1113s	- 0.0979s	1	1	Yes
complex.pl	3	0.0279s	0.4478s	- 0.4199s	2	2	Yes
complex2.pl	4	0.0258s	–	–	2	2	–[1]
complex3.pl	6	0.1211s	0.4271s	- 0.3060s	7	3	Yes
complex4.pl	6	0.1264s	1.9147s	- 1.7883s	7	3	Yes
complex5.pl	9	4.0087s	6.8243s	- 2.8156s	36	36	Yes
prop.pl	2	0.0129s	–	–	1	1	–[1]
prop2.pl	2	0.0194s	–	–	2	2	–[1]
self.pl	1	0.0027s	0.2232s	- 0.2205s	1	1	Yes
simple1.pl	3	0.0421s	–[2]	–	3	2	No
simple2.pl	2	0.0078s	–	–	1	1	–[1]
simple3.pl	3	0.0294s	–	–	2	2	–[1]
simple4.pl	2	0.0114s	–	–	1	1	–[1]
simple5.pl[3]	2	0.0090s	–[1]	–	1	1	–
twocoins.pl[3]	2	0.0095s	0.1303s	- 0.1208s	1	1	Yes

1: The example contained propagation rules.

2: The checker [1] detected the program wrongly as confluent.

3: Example also used in [1].

The examples test different cases to check the correctness of the CC^p. The examples show basic problems that are solved by both checkers.

The huge runtimes for the more complex examples come from the amount of comparisons, checks and printed out lines. A factor for the better runtimes of the CC^p is the lesser variable management because the CC^p only understand simple comparisons.

Another reason for the lower runtime is the less amount of redos. They require more runtime and memory management like the garbage collection. Since the CC^p is running with many CHR-Rules the possibilities for redos is smaller. This can be read out by the `profile/1` predicate in SWI-Prolog. For the `complex5.pl` example the checker [1] got more than 3.2 millions redos while the CC^p got around 85,000 redos. The lesser amount of redos is done by using CHR-rules and the lesser variable and guard management.

One more reason that the CC^p is faster on most of the examples is the lesser use of print outs on console. With increasing number of critical pairs the print out of the checker [1] is getting bigger than the one of the CC^p. Since the checker uses one `write/2` predicate to print out all results only the `nl/0` predicates can be compared for a fair comparison. While the CC^p uses around 0.06s for these predicates in the example `complex5.pl` the checker [1] uses around 0.18s for it. Another reason is the use of many CHR-constraints in the CC^p for checking the confluence. It can even check parts of itself with the result that `completion.pl` is confluent while `cp.pl` is not. The other parts are to big to check.

Also the garbage collection is a reason for the altering runtimes between both checkers. While CC^p uses around 0.02s for the garbage collection in the example `complex5.pl` the checker [1] uses around 0.16s for it.

The result of this comparison is that the runtimes of both checkers are similar to another while checking small programs but differ with bigger programs. In these cases the CC^p is faster. Both checkers still return the same correct result in all but one example. The example where the checker [1] detected the program wrongly as confluent was `simple1.pl`. In the example the following three rules were given:

Example 27. *simple1.pl:*

```
r1 @ a(X) <=> X<5 | b(X).
r2 @ a(X) <=> X>5 | c(X).
r3 @ a(X) <=> d(X).
```

The program has 3 non-trivial overlaps: (r1,r2), (r1,r3) and (r2,r3). Ther overlap between rule 1 and 2 is *not* critical since both rules cannot be applied at the same time thanks to the guards of the rules. The other overlaps *are* critical. If `X < 5` is satisfied rule 1 and 3 can be applied resulting into the state pair `(b(X),X<5),(d(X))` that is obviously not not joinable and thus critical. If `X > 5` is satisfied rule 2 and 3 can be applied and will result into the critical state pair `(c(X),X>5),(d(X))`. Since the program got at least one non-joinable state pair it is *not* confluent.

8.3 Tests with increasing complexity

The examples in this section get more and more complex with each step. They differ from one to four rules, confluent or not and with variables or not. The complexity of each examples can be identified by its name. The tests were done on the same conditions like the ones before.

Example	CC^p	Check[1]	CC^p - Checker Time	Confluent	Variables
`oneRule.pl`	0.0015s	0.0016s	- 0.0001s	Yes	No
`oneRuleVar.pl`	0.0016s	0.0017s	- 0.0001s	Yes	Yes
`twoRulesNc.pl`	0.0121s	0.0194s	- 0.0073s	No	No
`twoRulesVarNc.pl`	0.0077s	0.0256s	- 0.0179s	No	Yes
`twoRulesC.pl`	0.0011s	0.0018s	- 0.0007s	Yes	No
`twoRulesVarC.pl`	0.0012s	0.0018s	- 0.0006s	Yes	Yes
`threeRulesNc.pl`	0.0328s	0.0933s	- 0.0605s	No	No
`threeRulesVarNc.pl`	0.0311s	0.0959s	- 0.0648s	No	Yes
`threeRulesC.pl`	0.0011s	0.0014s	- 0.0003s	Yes	No
`threeRulesVarC.pl`	0.0013s	0.0017s	- 0.0004s	Yes	Yes
`fourRulesNc.pl`	0.0821s	0.5389s	- 0.4568s	No	No
`fourRulesVarNc.pl`	0.0792s	0.5593s	- 0.4801s	No	Yes
`fourRulesC.pl`	0.0013s	0.0018s	- 0.0005s	Yes	No
`fourRulesVarC.pl`	0.0015s	0.0020s	- 0.0005s	Yes	Yes
`eighteenRulesNc.pl`	22.2251s	20.0626s	+ 2.1625s	Yes	No

In each example both checkers returned the same result and detected the (non-)confluence correctly.

What can be seen in these examples is that both checkers do not differ in the runtime of simple confluent examples. In that cases the runtime of them is nearly identical. The difference between the usage of variables or none is minor in these cases.

When checking non-confluent small examples the CC^p is faster than the checker [1]. There is no significant difference between the usage of variables or none. In the last example the CC^p is slower than the checker [1]. This examples was the maximum of rules and overlaps that the CC^p could handle in this test environment. Added only one more rule to the program resulted into a out of global stack exception of SWI-Prolog. The example contained eighteen rules with a total of 153 overlaps and critical pairs.

The checker [1] can handle such extreme examples better because in the CC^p it is very expensive to check rules for overlaps. Only simplification rules were used in the example, for each of the rule pairs (153) a `pair/2` constraint is generated. Since SWI-Prolog calls CHR-Rules in their order of the source file the rules that got `pair/2` as the head of the rule is called 459 times. Since the `single_matching/2` predicate is in all guards of this rules it is 1224 times. In each of this calls the `matching/3` predicate is called. It is called by two lists with one member (`[a],[a]`). In every call lists get defined and declared, unification and compound checks are

done, constraints are split into functor and arguments and more. All these action need some amount of memory so much that the garbage collection spent about 12.3s of the 22.2251s computation time.

8.4 Tests (Completion)

Since the checker written by Johannes Langbein [1] does not support the completion algorithm no comparison can be made. The tests are made on the same machine like the confluence checks. All examples can be found in the folder of the CC^p.

Example	Rules	Runtime	Completable	Generated Rules
cell.pl	1	0.0046s	No	–
coin.pl	2	0.0407s	No	–
complex.pl	3	0.8312s	Yes	–
complex2.pl	4	∞	Unknown[1,4]	∞
complex3.pl	6	∞	Unknown[1]	?
complex4.pl	6	∞	Unknown[1]	?
complex5.pl	9	∞	Unknown[1]	?
prop.pl	2	0.0536s	Yes	1
prop2.pl	2	0.0784s	No[2]	1
self.pl	1	0.0967s	No[2]	–
simple.pl	2	0.1871s	Yes	1
simple1.pl	3	0.2794s	Yes	4
simple2.pl	2	0.0464s	Yes	1
simple3.pl	3	0.1271s	Yes	2[3]
simple4.pl	2	∞	No[4]	1[4]
simple5.pl	2	0.0653s	Yes	1
twocoins.pl	2	0.0455s	No	–

1: The calculation was not finished before SWI-Prolog ran out of global stack (set to 1GB).

2: Self-overlaps cannot be fixed by the completion algorithm.

3: The algorithm generated duplicate rules.

4: The completion algorithm created an infinite loop.

Since the completion algorithm is semi-decidable it will not complete for every input program. A program can still be completable when SWI-Prolog runs out of global stack. A self-overlap cannot be fixed because a critical pair like `(Coin = head), (Coin = tail)` cannot be resolved by CHR-Rules. The rule would have to express `(head = tail)` this cannot be done by definition of predicate logic since constants must not be unified.

With the testcases it can be seen that the CC^p is calculating correctly. All examples are completed with the correct new set of rules that are generated by the completion algorithm.

9 Conclusion and Future Work

A new correct confluence checker for Constraint Handling Rules that supports propagation rules and completion has been implemented. In this section a short overview and conclusion for all that was done will be given and a talk about what could be implemented and improved in the future.

9.1 Conclusion

The CC^p implements the theoretical concept described in chapter 3.1. It can understand all types of CHR-Rules and can check CHR-Programs for confluence. For the transforming of the input program and the checking Prolog and CHR were used. To check to confluence a CHR constraint solver was implemented that supports all types of rules with and without guards. The main difference between the older confluence checkers like [1] and the one described in this thesis is that the CC^p supports propagation rules. With this new ability the completion algorithm can be implemented.

A program may be completable if it is non-confluent, at least one of states of the critical pair contains constraints and the built-ins of these both states are not a contradiction. The completion algorithm that is used will produce new rules that can lead to the confluence of the input program. But since the completion algorithm is not decidable it can also lead to an infinite loop or no new rules at all. The current implementation has a basic loop detection that does not generate rules where the head and the body are the same or the head is a part of the body. The rules that get newly created are written into an output file together with the rules and overheads of the original input program.

The whole program has a basic understanding of built-ins and supports basic relations. It can decide if two guards can be fired at the same time or not for these built-ins. A problem with the built-ins is, that the program would have to store information about variables to do more advanced guard checks. This would be needed to support more built-ins. The completion algorithm is not decidable therefore the computation can be infinite.

In the evaluation the performance of the CC^p was checked to detect potential and the limits of the implementation. It was shown that the CC^p provides the same results like the checker checker for the given examples.

9.2 Future Work

In the current implementation of the confluence checker not many built-ins that are supported. This could be addressed by a complex constraint solver that has its own variable store where information for all variables is gathered and used to solve guards or other built-ins. With these informations the built-ins could be called and potentially solved. Frank Richter implemented such a support for the old confluence checker in oeqck. The built-in support for the CC^p can be build onto the concept of this work.

To improve the performance and reduce redundancies the parallelism in the completion algorithm can be improved. In the current implementation every critical pair calls the completion on its own. By adding a program that overwatches the whole program and collects all data after each step of the program the runtime of the program could be improved.

The confluence check could be further enhanced by checking all overlaps, collecting the results of them and evaluate the complete result. By this change the completion could be made linear and no duplicate rules would be created. Another possibility would be a global rule list constraint which would be altered every time a new rule is created. With this the rule list would always be up-to-date and the creation of duplicate rule would be prevented. This could eliminate the checking of already fixed overlaps and improve the overall performance of the checker.

Bottlenecks are a problem when checking programs with many overlaps. The generation of the overlaps is such a bottleneck in the CC^p, every rule gets matched with every other. In that step the constraint store is filled with `pair/2` constrains. The check if a pair has an overlap or not is started after every pair has been generated. With huge programs the memory usage of the pairs is huge. This could be resolved by generating pairs only if there is an overlap.

The completion part of the program could also be enhanced for a better user experience. More loop detection can be done by adding more predicates to do this. Cases where two rules generate constraints to fire the other rule and so on could be detected by this.

A Disk Content

Folder ’./CCp/Code’

This folder contains the confluence checker for CHR programs with propagation rules and the implementation of the completion algorithm.

Folder ’./CCp/Examples/Confluent’

This folder contains the confluent examples used in this thesis.

Folder ’./CCp/Examples/Non-Confluent’

This folder contains the non-confluent examples used in this thesis.

Folder ’./CCp/Examples/Complexity Examples’

This folder contains the examples with raising complexity from the evaluation chapter.

Folder ’./CCp/Bachelor-Thesis-Daniel-Bebber.pdf’

This file is the pdf version of this work.

Code Listings

List of Figures

Bibliography

[1] *State Equivalence and Confluence Checker*. `http://www.uni-ulm.de/en/in/pm/research/topics/chr/info/downloads.html,` . – Accessed: 2015-10-15

[2] ABDENNADHER, Slim: Operational semantics and confluence of constraint propagation rules. In: *Third International Conference on Principles and Practice of Constraint Programming* (1997), S. 252–266

[3] ABDENNADHER, Slim ; FRUEHWIRTH, Thom: On Completion of constraint handling rules. In: *Fifth International Conference on Principles and Practice of Constraint Programming* (1998), S. 25–39

[4] ABDENNADHER, Slim ; FRUEHWIRTH, Thom ; MEUSS, Holger: On Confluence of Constraint Handling Rules. In: *Second International Conference on Principles and Practice of Constraint Programming* (1996), S. 1–15

[5] ABDENNADHER, Slim ; FRUEHWIRTH, Thom ; MEUSS, Holger: Confluence and semantics of constraint simplification rules. In: *Constraints* (1999), S. 133–165

[6] FRUEHWIRTH, Thom: *Constraint Handling Rules*. Cambridge University Press, 2009

[7] FRUEWIRTH, Thom: Introducing Simplification Rules. In: *Workshop Logisches Programmieren* (1991)

[8] LANGBEIN, Johannes ; RAISER, Frank ; FRUEHWIRTH, Thom: *A State Equivalence and Confluence Checker for CHR*, Ulm University, Germany, Diplomarbeit, 2010

[9] RAISER, Frank ; BETZ, Hariolf ; FRUEHWIRTH, Thom: Equivalence of CHR states revisited. In: *6th International Workshop on Constraint Handling Rules (CHR)* (2009), S. 34–48

[10] RICHTER, Frank: *An Operational Equivalence Checker for CHR*, Ulm University, Germany, Diplomarbeit, 2014

[11] SNEYERS, Jon ; VAN WEERT, Peter ; SCHRIJVERS, Tom ; DE KONINCK, Leslie: As time goes by: Constraint Handling Rules. In: *TPLP* (2010), Nr. 1, S. 1–47

Name: Daniel Martin Yaspar Bebber Matrikelnummer: 780328

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Daniel Martin Yaspar Bebber