ulm university universität
uulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Programmiermethodik und Compilerbau

An Operational Equivalence Checker for CHR

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Frank Richter
frank-1.richter@uni-ulm.de

Gutachter:
Prof. Dr. Thom Frühwirth

Betreuer:
M.Sc. Daniel Gall

2014

Version July 22, 2014

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

There are many occasions where it is interesting to know if two programs are equivalent. Based on the strict definition of operational equivalence, this work presents a tool to check two CHR programs for equivalence. It uses the state equivalence and confluence checker written by Johannes Langbein. This offers a tool to analyze simple CHR programs for confluence and operational equivalence. Furthermore support for more built-ins is added to the confluence and operational equivalence checker, to remove some of its limitations.

Contents

1 Introduction

In programming language semantics determining when two programs should be considered equivalent is a fundamental and difficult question. [2] In the following a strict notion of program equivalence is used. If the computations of two given programs result in the same answer for any given query, they are considered operational equivalent. For this there is a decidable, sufficient and necessary syntactic condition of CHR programs, as long as they are terminating and confluent.[4, p. 128]

1.1 Motivation

There are many cases where knowing if two programs can be considered equivalent is interesting and important. For example to check the correctness of program transformations or to see if different modules or libraries with similar functionality have program parts that are equivalent [2], so a program that could check if two programs are equivalent would be useful.

1.2 Aim

The aim of this work is to create a program, that can check if simple CHR programs are equivalent. This work will benefit from the rather unique property of having a decidable test for operational equivalence that CHR offers [4, p. 128]. The main part of the operational equivalence checker will be written in Prolog except for some cases where CHR is used. It is based on the confluence checker written by Johannes Langbein [6]. The first step is a basic implementation of the definition of operational equivalence. Then some limitations concerning Prolog built-ins are removed by representing those built-ins with CHR constraints and constraint solvers. Support for those built-ins then is added to the confluence checker [6] and finally the confluence checker and the operational equivalence checker are united to a program that is able to check two programs for confluence and subsequent for operational equivalence, given they passed the confluence check.

2 Constraint Handling Rules

Constraint Handling Rules (CHR) was invented in 1991 by Prof. Dr. Thom Frühwirth [5]. It 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 [4, p. xvii].

CHR always needs to be provided with data types and predefined constraints from its host language H. The host language is denoted in round brackets and needs to provide at least the constraints $true$ and $fail$, and syntactic equality and inequality checks. CHR(H) denotes CHR embedded in the host language H. There are several implementations like CHR($Java$), CHR($Haskell$) and CHR(C) but CHR($Prolog$) is the most common one. [8]

This chapter starts with a brief presentation of the syntax and the semantics of CHR. Based on this State Equivalence will be defined what is fundamental for the rest of this chapter, where Confluence in CHR and Operational Equivalence in CHR will be explained.

2.1 Syntax

As a first-order logic language, CHR consists of a set of variables V, a set of function symbols Σ, and a set of predicate symbols Π. Function and predicate symbols have an associated arity like the number of arguments they take. A functor is a symbol f with the arity n written in the notation f/n. If a function symbol has the arity zero it is called a constant while a predicate symbol with arity zero is called proposition. A constraint is a distinguished predicate of first-order logic. There are two types of constraints, the pre-defined built-in constraints, which are provided by the host language and the user-defined CHR constraints.[4, p. 53]

Definition 1. A CHR Program consists of a finite set of rules. There are three types of rules: simplification-, propagation- and simpagation rules. The form of those rules can be seen in Figure 2.1. A rule has an optional name r that is separated with an $@$ from the actual rule. The rule-name is an optional, unique identifier. Each type of rule has a head that may not be empty and consists of CHR constraints, a guard that may be empty and consists of built-in constraints and a body that may not be empty and can have built-in constraints as well as CHR constraints. The exact syntax can be seen in Figure 2.1.

$Built-in\ constraint:$	C, D	::= $c(t_1, ..., t_n) \mid C \wedge D, \quad n \geq 0$
$CHR\ constraint:$	E, F	::= $e(t_1, ..., t_n) \mid E \wedge F, \quad n \geq 0$
$Goal:$	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	::= $r@E_1 \backslash E_2 \Leftrightarrow C \mid G$
$CHR\ rule:$	R	::= $SR \mid PR \mid SPR$
$CHR\ program:$	P	::= $R_1 ... R_m, \quad m \geq 0$

Figure 2.1: Abstract syntax of CHR programs and rules [4, p. 54]

2.2 Semantics

This section will describe the operational semantics of CHR. Only the very abstract and the abstract semantics will be described here, because they are sufficient to explain confluence and operational equivalence.

In the following P will be a CHR program and CT will be a constraint theory for the built-in constraints. A rule that fires will add the constraints of its body to the constraint store. A simplification rule removes the head constraints from the constraint store while a propagation rule keeps the head constraints. A simpagation rule has head constraints that are removed as well as head constraints that are kept.

2.2.1 Very Abstract Semantics

The very abstract operational semantics of CHR is given by a nondeterministic state transition system. [4, 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. [4, p. 56]

For the transitions we use rules in the head normal form (HNF). 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. A transition represents a rule application. The formal definition of the transition relation can be seen in Figure 2.2 The upper-case letters H_1, H_2, C, B and G represent conjunctions of constraints that can be empty. What happens if an applicable rule is applied, is that the CHR constraints H_1 are kept while the CHR constraints H_2 are removed. The resulting state additionally consists of the the guard C and the body B. [4, p. 56]

This transition system is nondeterministic, because if several rules are applicable one is chosen nondeterministically and this choice cannot be undone. [4, p. 56]

Apply
$(H_1 \wedge H_2 \wedge G) \mapsto_r (H_1 \wedge C \wedge B \wedge G)$
if there is an instance with new local variables $\overline{x}$ of a rule named r in P.
$r@H_1 \backslash H_2 \Leftrightarrow C|B$
and $CT \models \forall(G \rightarrow \exists \overline{x} C)$

Figure 2.2: Transition of the very abstract operational semantics of CHR [4, p. 56]

2.2.2 Abstract Semantics ω_t

The very abstract semantics do not pay much attention to termination. Especially propagation rules can easily lead to nontermination, because the application of a propagation rule dose not make the rule inapplicable and a failed state makes any rule applicable. These issues are addressed by the abstract operational semantics of CHR, which add a distinction between processed and unprocessed constraints. [4, p. 59]

Since any rule is applicable in a failed state and it can only lead to another failed state, all failed states are declared final states, to prevent the trivial nontermination of failed states. To avoid the repeated application of propagation rules it is made sure of that they are not applied more than one time to the same constraints. The information of all CHR constraints to which propagation rules have been applied is stored in the so-called propagation history. [4, p. 60]

Like in the very abstract semantics, the guard is a conjunction, but head and body are now multisets of atomic constraints. Each constrain now has a unique identifier (which is a natural number). For a CHR constraint c with the identifier i the notation $c\#i$ is used, such a constraint is called a numbered constraint. In the following 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. [4, p. 60]

The following definition for states in the abstract semantics is given in [4, p. 60]:

Definition 3. 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 which contains all 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.

The following definition for failed states and final states is taken from [4, p. 61]:

Definition 4. Given an initial goal (query, problem, call) G with variables V, the initial state is $\langle G, \emptyset, true, \emptyset \rangle_1^V$.
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 is either a successful state where no transition is possible anymore or a failed state. Given a final state $\langle G, S, B, T \rangle_n^V$, its conditional or qualified) answer (solution, result) is the conjunction $\exists \overline{y}(chr(S) \wedge B)$, where $\overline{y}$ are the variables not in V.

Figure 2.3 shows the transition rules for the abstract operational semantics ω_t. The solve transition uses the built-in solver to add a built-in constraint from the goal G to the built-in constraint store B. The resulting built-in store is simplified by an unspecified amount, in the worst case it is just the original conjunction of the new constraint with the old built-in store. In the introduce transition a CHR constraint is added to the CHR store S, gets the next free integer n as identifier and the next free integer is set to $n + 1$. The apply transition picks a rule r from the program P and applies (fires, executes) it. The criterion for a picked rule is that constraints matching its head exist in the CHR constraint store S and that its guard g is logically implied by the built-in store B under the matching. [4, 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 $\overline{x}$ of the form

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

where $CT \models \exists(B) \ \wedge\ \forall(B \rightarrow \exists \overline{x}(chr(H_1) = H_1' \wedge chr(H_2) = H_2' \wedge g))$ and $(r, id(H_1) + id(H_2)) \notin T$.

Figure 2.3: Transition of the abstract operational semantics ω_t [4, p. 63]

2.3 State Equivalence

This section uses the following definition for states that can be found in [7]:

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 strictly local variable.

With the definition of states it would be possible to use the logical equivalence when talking about state equivalence, but when thinking of all failed states as equivalent and considering the multiset character of CHR constraints, a slightly stricter notion of equivalence is preferable. The basic idea is, that if states are equivalent, then the same rules should be applicable to those states. [4, p. 71]

Definition 6 (State Equivalence)**.** Equivalence between CHR states is the smallest equivalence relation $\equiv$ over CHR states that satisfies the following conditions [7] :

1. (Equality as Substitution)

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

2. (Transformation of the Constraint Store)
   If $CT \models \exists \overline{s}.\mathbb{B} \leftrightarrow \exists \overline{s}'.\mathbb{B}'$ where $\overline{s}, \overline{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$ respectively, then:

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

3. (Omission of Non-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 Theorem 7 [7]:

Theorem 7 (Criterion for $\equiv$)**.** *Let* $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \sigma' = \langle \mathbb{G}', \mathbb{B}', \mathbb{V} \rangle$ *be CHR states with local variables* $\overline{y}, \overline{y}'$ *that have been renamed apart.*

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

Since Theorem 7 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 pose a problem. [7]

2.4 Minimal States

During program analysis there is normally an infinite amount of possible states. For analysis, it is necessary to limit the amount of states to a finite amount of so-called minimal states. Every rule has a smallest, most general state that allows it to fire. [4, p. 101]

Definition 8 (Minimal State)**.** The minimal state of a rule is the conjunction of the head and the guard of the rule. [4, p. 101]

If any constraint from a minimal state is removed, the rule would no longer be able to fire, while adding constraints cannot prevent the possibility of the rule to fire. Every state that allows a rule to fire contains the minimal state of the rule. [4, p. 101]

2.5 Joinability

Another interesting property of two states is, if they can result in equal states. This property is called joinability.

Definition 9 (Joinability)**.** Two states S_1 and S_2 are joinable if there exist states S_1', S_2' such that $S_1 \mapsto^* S_1'$ and $S_2 \mapsto^* S_2'$ and $S_1' \equiv S_2'$. [4, p. 102]

2.6 Confluence

Confluence is a property of a program that guarantees the same final state for any computation of a goal independent of which applicable rules are fired. This means that if a program is confluent, the order of rules in the program and the order of constraints in a goal does not matter. [4, p. 101]

Definition 10 (Confluence)**.** A CHR program is confluent if for all states S, S_1, S_2

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

Definition 10 is illustrated in Figure 2.1. [4, p. 102]

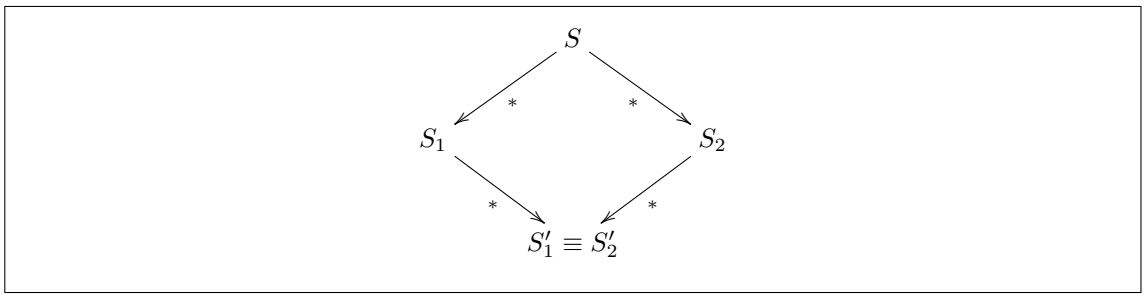

Figure 2.1: Confluence diagram [4, p. 103]

2.6.1 Test for confluence

Since in general, there is an infinite amount of possible states, it is not possible to simply check every of those states for joinability. For a terminating program it is possible to limit the states for the joinability test to a finite number of most general states, the so-called overlaps. These overlaps are states in which more than one rule is able to fire. Those two rules can actually be the same rule with a different order of head constraints or at least one different head constraint. By merging the minimal states of two rules and equating at least one head constraint from one rule with one from the other rule the two rules are overlapped. The two states resulting from applying both rules to an overlap are called critical pair. If any critical pair of a program is not joinable, the program itself is not confluent. [4, p. 102]

Definition 11. Let R_1 be a simplification or simpagation rule and R_2 be a (not necessarily different) rule, whose variables have been renamed apart. Let $H_i \wedge A_i$ be the 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. [4, p. 103]

For terminating CHR programs, a decidable, sufficient and necessary condition for confluence is given by the following theorem 12 [4, p. 104]

Theorem 12. *A terminating CHR program is confluent iff al its critical pairs are joinable. [4, p. 104]*

Up to this point this section basically just covered the very abstract semantics (see 2.2.1). When considering the abstract semantics (see 2.2.2) one has to take the propagation history into account. For each state of a critical pair, the propagation history needs to be set in a way, that no propagation rule can fire if it would only use constraints that have been present in the overlap. This means, that an overlap $S \wedge B$, where S are the CHR constraints and B are the built-in constrains is associated with the ω_t state $\langle \emptyset, S', B, \emptyset \rangle_n^V$, where S' are n consistently numbered CHR constraints, so that $S = chr(S')$, and V contains all variables of the overlap. In such a ω_t state of a critical pair, the propagation histories are set to $prop(S')$, where $prop(S')$ is a function that returns a propagation history with an entry for each propagation rule of the program for each valid combination of constraints from S'. [4, p. 108]

2.6.2 Examples

This section will give some example CHR programs that are tested for confluence. Those examples are taken from [4]. A CHR program that consists either only of propagation rules or single headed simplifcation rules that do not overlap are trivial cases that are obviously confluent. [4, p. 105]

Example 13. *Consider the following Coin-toss program:*

$$throw(Coin) \Leftrightarrow Coin = head.$$
$$throw(Coin) \Leftrightarrow Coin = tail.$$

This only has the overlap:

$$throw(Coin)$$

what leads to the critical pair:

$$(Coin = head, Coin = tail)$$

Those are two final states that are different and thus not joinable. So the program is not confluent.

Example 14. *Consider the following single rule program:*

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

This rule has the overlaps:

$$p(X) \wedge q(Y_1) \wedge q(Y_2)$$
$$p(X_1) \wedge p(X_2) \wedge q(Y)$$

which lead to the critical pairs:

$$(q(Y_1), q(Y_2))$$
$$(p(X_1), p(X_2))$$

Both critical pairs consist of two final states that are different and thus not joinable. So the program is not confluent.

Example 15. *Consider the following CHR program with a propagation rule:*

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

It has the overlap:

$$r \wedge p \wedge q$$

Figure 2.2 shows that it is important to add the propagation histories, because without it would always end in the final state $p \wedge q$ (as seen in (1)), while with the added propagation history for rule $r1$ it results in the not joinable critical pair $(p, p \wedge q)$ (as seen in (2)).

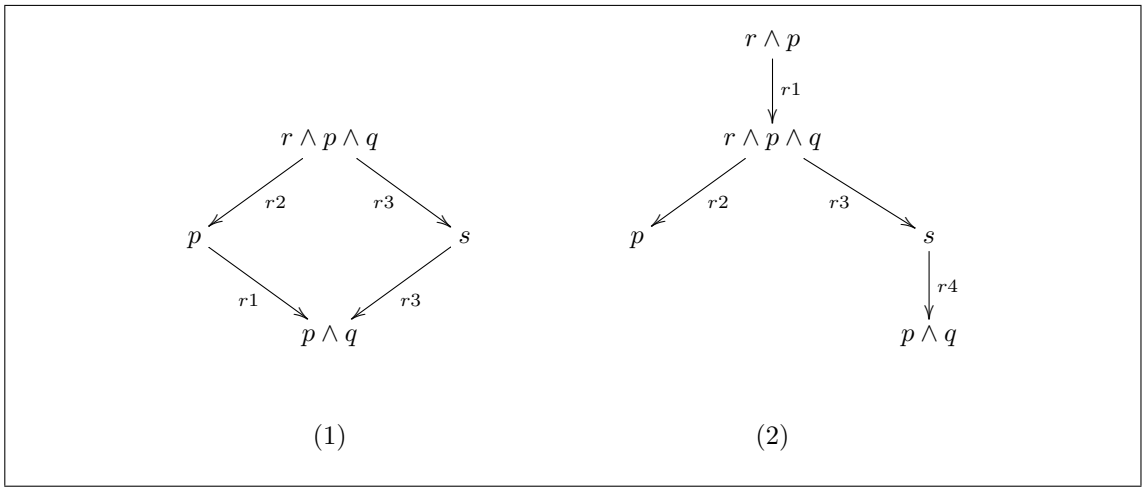

Figure 2.2: Violated confluence in larger state [4, p. 109]

Example 16. *Consider the following CHR program for less or equal:*

$$\begin{array}{lrl} duplicate \ @ & (X \leq Y) \ \wedge \ (X \leq Y) & \Leftrightarrow \quad (X \leq Y) \ . \\ reflexivity \ @ & (X \leq X) & \Leftrightarrow \quad true. \\ antisymmetry \ @ & (X \leq Y) \ \wedge \ (Y \leq X) & \Leftrightarrow \quad X = Y \ . \\ transitivity \ @ & (X \leq Y) \ \wedge \ (Y \leq Z) & \Rightarrow \quad (X \leq Z). \end{array}$$

Figure 2.3 shows the overlap for reflexivity and antisymmetry and shows that the resulting critical pair is joinable.

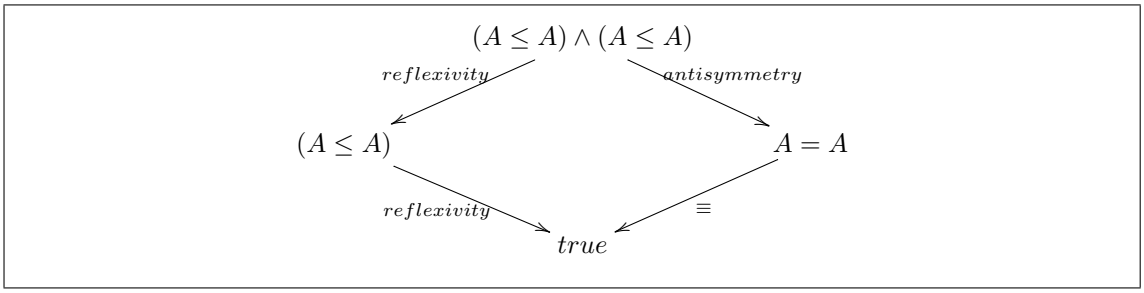

Figure 2.3: Joinable overlap of reflexivity and antisymmetry rules [4, p. 107]

Another overlap can occur with the transitivity rule and the antisymmetry rule, e.g. :

$$(X \leq Y) \wedge (Y \leq Z) \wedge (Y \leq X)$$

This overlap leads to the following critical pair:

$$((X \leq Y) \wedge (Y \leq X) \wedge (Y \leq Z) \wedge (X \leq Z), X = Y, (X \leq Z))$$

By taking the first state of this critical pair and applying the antisymmetry rule and the duplicate rule one gets a state that is equivalent to the second state.

$$\begin{aligned} & (X \leq Y) \wedge (Y \leq X) \wedge (Y \leq Z) \wedge (X \leq Z) \\ \mapsto_{antisymmetry} \quad & (Y \leq Z) \wedge (X \leq Z) \wedge X = Y \\ \mapsto_{duplicate} \quad & (X \leq Z) \wedge X = Y \end{aligned}$$

This shows that it is also a joinable critical pair.

2.7 Operational Equivalence

This section gives the definitions the operational equivalence checker (see section 4) is based and shows some example CHR programs that are checked for operational equivalence with the help of those definitions.

2.7.1 Definition

Operational equivalence of two programs is the fact that for any given (initial) state equivalent states can be reached by the computation in each program. [4, p. 128] This leads to the following definition that is taken from [4, p. 128] :

Definition 17 (Operational Equivalence)**.** Let the notation $\rightarrow_p$ denote a transition using program P.

Two CHR programs P_1 and P_2 are operationally equivalent if all states are P_1, P_2-joinable.

A state S is P_1, P_2-joinable iff there are computations $S \rightarrow^+_{P_1} S_1$ and $S \rightarrow^+_{P_2} S_2$ such that $S_1 \equiv S_2$ or S is a final state in both programs.

2.7.2 Test for Operational Equivalence

For CHR programs, that are confluent and terminating, so-called well behaved programs, there is a straight forward test for operational equivalence that is given in the following theorem which is taken from [4, p. 128]:

Theorem 18. *Two well-behaved programs P_1 and P_2 are operationally equivalent iff all minimal states of the rules in P_1 and P_2 are P_1, P_2-joinable.*

2.7.3 Examples

Example 19. *Consider the following two programs to determine the minimum:*

P_1 :

$$\begin{array}{lll} r_1 \ @ & min(X) \ \wedge \ min(Y) \ \Leftrightarrow & X < Y \mid min(X). \\ r_2 \ @ & min(X) \ \wedge \ min(Y) \ \Leftrightarrow & X = Y \mid min(X). \end{array}$$

P_2 :

$$r_1 \ @ \qquad min(X) \ \wedge \ min(Y) \ \Leftrightarrow \qquad X \leq Y \mid min(X).$$

P_1 has two minimal states and P_2 has one minimal state, so the list of states that needs to be checked is:

$$(min(X) \wedge min(Y) \wedge X < Y \ , \ min(X) \wedge min(Y) \wedge X = Y \ , \ min(X) \wedge min(Y) \wedge X \leq Y)$$

For the first two minimal states the computation of P_1 and P_2 results in equivalent final states.

$$\begin{array}{ll} min(X) \wedge min(Y) \wedge X < Y & \mapsto_{P_1 r_1} \ min(X) \\ min(X) \wedge min(Y) \wedge X < Y & \mapsto_{P_2 r_1} \ min(X) \end{array}$$

$$\begin{array}{ll} min(X) \wedge min(Y) \wedge X = Y & \mapsto_{P_1 r_2} \ min(X) \\ min(X) \wedge min(Y) \wedge X = Y & \mapsto_{P_2 r_1} \ min(X) \end{array}$$

But the third minimal state already is a final state for P_1 while P_2 reaches a not equivalent final state:

$$min(X) \wedge min(Y) \wedge X \leq Y \qquad \mapsto_{P_2 r_1} min(X)$$

This means that the two programs are not operational equivalent even so they are logically equivalent.

Example 20. *Consider the following two "hello world" programs:*

P_1 :

$$\begin{array}{lll} r_1 \ @ & a \ \Leftrightarrow & helloworld. \\ r_2 \ @ & b \ \Leftrightarrow & helloworld. \end{array}$$

P_2 :

$$\begin{array}{lll} r_1 \ @ & a \ \Leftrightarrow & b. \\ r_2 \ @ & b \ \Leftrightarrow & helloworld. \end{array}$$

Both programs have equivalent minimal states, after removing the duplicates the list of minimal states is:

$$(a, b)$$

For both minimal states the computation of P_1 and P_2 leads to equivalent final states.

$$\begin{array}{llll} a & & \mapsto_{P_1 r_1} & helloworld \\ a & \mapsto_{P_2 r_1} \ b & \mapsto_{P_2 r_2} & helloworld \\ & & & \\ b & & \mapsto_{P_1 r_2} & helloworld \\ b & & \mapsto_{P_2 r_2} & helloworld \end{array}$$

This means the two programs are operational equivalent.

3 The State Equivalence and Confluence Checker

This section describes the state equivalence and confluence checker [1] the operational equivalence checker is based on. It is written in SWI Prolog and organized in different modules.

3.1 CHR Parser

The CHR parser is located in the module `chrparser.pl`. It expects a syntactically correct file and does not provide any error handling. Everything except for CHR simplification and simpagation rules and the `chr\_constraint` directive that defines the CHR constraints with their arity, is ignored by the parser. [1]

The parser is used by calling the predicate read_rules/3 which reads the rules and the list of CHR constraints from a file. It is called with

$$read\_rules(+FileName, -Rules, -CHRC)$$

where $FileName$ is a string with the path to the file that is parsed, $Rules$ is a list with all simpagation and simplification rules and $CHRC$ is the list that contains all CHR constraints that appear in the program. [1]

The $Rules$ list represents rules with a $rule(S, KH, RH, G, B)$ term, where S is the rule as a string as it is represented in the source file and KH, RH, G and B are lists that represent the kept head constraints, the removed head constraints, the guard and the body of the rule. KH and G may be empty.[1]

(Note: The $+$ infront of an argument means that the predicate takes this argument as input while a $-$ says that this is returned by the predicat.)

3.2 State Equivalence

In the module `stateequiv.pl` Theorem 7 is implemented for the equivalence relation $\equiv$ over CHR states. States follow Definition 5 and are represented as Prolog terms. Those terms have the form $state(G, B, V)$ where G is a list representing the goal store, B is a list representing the built-in store and V is a list for the global variables of the state. The CHR constraints are represented as Prolog terms. The built ins are represented as $=/2$, $true$ and $false$ and there are

no other built-ins supported. The variables are represented as Prolog variables. An empty goal or builtin store or an empty set of variables is represented by an empty list. [6]

A check for state equivalence is performed by a call of the predicate

$$equivalent\_states(+S1, +S2)$$

where $S1$ and $S2$ are states represented by Prolog terms as described earlier and the predicate succeeds if the two states are equivalent. [1]

3.3 Confluence

The module `conflcheck.pl` is an implementation of the criterion for confluence from Theorem 12. First all possible critical pairs of all rules in the program are created. They are checked for joinability by calling the CHR program with both states separately as query and checking the two resulting states for equivalence. The equivalence check is performed with $stateequiv/2$ from the `stateequiv.pl` module that is described in section 3.2. [6]

A check for confluence is performed by a call of the predicate

$$check\_confluence(+FileName)$$

where $FileName$ is a String with the path to the file that is checked for confluence. The predicate succeeds regardless if the CHR program is confluent or not. All non-joinable critical pairs will be printed on the screen. [1]

3.4 Limitations

In Section 2.6.1 it is described that the propagation history for the constraints of the critical pair needs to be added to prevent propagation rules from firing twice. Since the confluence checker does not consider the propagation history it cannot support propagation rules.[1]

There is only support for the built-ins $=/2$ and $true$. The reason for this limitation is, that a most general CHR state needs to be able to have unbound variables with the information that certain entailment checks succeed, while Prolog in general requires the arguments of built-ins used in an entailment check to be ground.[6]

Another minor limitation is that no syntax check is performed on the program that is tested. Thus the confluence checker should only be called with valid CHR programs. [1]

4 The Operational Equivalence Checker

This section describes how the operational equivalence checker is implemented. First a basic implementation of Theorem 18 based on the state equivalence and confluence checker (see section 3) is shown. Since this implementation has the same limitations (see section 3.4) as the state equivalence and confluence checker, the rest of this section explains how support for additional built-ins has been realized to remove some of those limitations.

4.1 Basic Implementation of the Theorem

Since the state equivalence and confluence checker (see section 3) already offers a predicate to check for state equivalence (see section 3.2) and a CHR parser (see section 3.1) that extracts all rules and CHR constraints from a CHR program, it is used as basis for the operational equivalence checker.

4.1.1 Implementation

To implement an operational equivalence checker the module opeq.pl is created and has the predicate

$$opeq(File1, File2)$$

where $File1$ and $File2$ are Strings with the path to the two files that are checked for operational equivalence.

The first step of this predicate is to call $read\_rules(+FileName, -Rules, -CHRC)$ from the CHR parser (see 3.1) with both programs to get a list with all rules and a list with all constraints for each program. Next a check on the defined constraints is done with a call of:

$$check\_Chrc(+Chrc1, +Chrc2, -N)$$

where $Chrc1$ and $Chrc2$ are the lists of CHR constraints that are defined in program one and two. The predicate always succeeds. If $Chrc1$ and $Chrc2$ consist of the same elements (not necessarily in the same order) it returns $N = 0$ and if at least one element of one list does not occur in the other list it returns $N = 1$. This is check is mandatory, because two programs can obviously not be operationally equivalent if there exists a query that would cause one of them to throw an error that does not occur in the other program.

Theorem 18 says that the minimal states of all rules from both programs need to be checked, so the next step is to get a list with all minimal states. Therefor the following predicate is called:

$$list\_of\_min\_states(+Prog1, +Prog2, -Min\_States)$$

where $Prog1$ and $Prog2$ are the lists of rules for each program that the predicate $read\_rules/3$ (see Section 3.1) has returned. Min_States is a list of minimal states represented as Prolog terms with the form $minState(G, B, V)$ similar to the $state(G, B, V)$ term that the state equivalence checker uses (see section 3.2). The goal store G for such a minimal state for a rule is created by appending the list representing the removed heads to the list with the kept heads. The built-in store B is equivalent to the list that is representing the guard of the rule. The set of variables V can be determined by appending G and B and using the predicate $term\_variables/2$.

The next step is to call each program with each minimal state and check the resulting final states for equivalence. This is done in a similar way like it is done in the $states\_joinable/3$ `conflcheck.pl` [1]. But instead of one program that is consulted with two states, there need to be two programs that are consulted with one equivalent state. Since consulting two CHR programs that define constraints with the same name in the same module could be problematic, the modules `consult1.pl` and `consult2.pl` are used. After both programs have been called with a minimal state the resulting final states are checked for equivalence by calling:

$$eq\_states(+Result1, +Result2, -N, +S)$$

where $Result1$ and $Result2$ are the resulting final states of both programs and S is the minimal state that is currently tested. To check for state equivalence the predicate $equivalent\_states/2$ (see section 3.2) is called with $Result1$ and $Result2$ and if it succeeds $N = 0$ is returned. Otherwise $N = 1$ is returned and the minimal state is printed to the screen together with the remark, that it leads to not equivalent final states.

The N from each call of $eq\_states/4$ are added up. If all minimal states have been checked and the sum is zero, the two programs are operationally equivalent.

4.1.2 Limitations

Since this implementation so far basically just uses what the state equivalence and confluence checker offers, it has very similar limitations.

The parser does not parse propagation rules, so obviously programs with propagation rules cannot be checked, even though propagation rules would not cause problems in the check for operational equivalence, since in the check for operational equivalence the minimal states that are called are always initial states with an empty propagation history. The modified version of the parser that will be introduced later on has this limitation removed.

Just like in the state equivalence and confluence checker the only supported the built-ins are $=/2$ and $true$. The reason for this limitation is, that a most general CHR state needs to be able to have unbound variables with the information that certain entailment checks succeed, while Prolog in general requires the arguments of built-ins used in an entailment check to be ground. The next section describes how support for additional built-ins can be added by representing them with constraint solvers.

This implementation of an operational equivalence checker should only be called with valid and well-behaved CHR programs.

4.2 Adding Built-ins

The support of only three built-ins is a big limitation. The reason for those limitations is that Prolog cannot store information for unbound variables to let certain entailment tests with prolog built-ins succeed.

The basic idea for adding support for more built-ins is to represent those built-ins with constraint solvers so that the information for a succeeding entailment check can be stored in a CHR constraint where unbound variables pose no problem. First this section will take a look at CHRat which is a modular version of CHR that allows CHR constraints to be used in the guard of a rule and shows a way to represent entailment checks for CHR constraints. Then the way further built-ins for the operational equivalence checker are implemented is explained and the constraint solvers for the new built-ins are shown. Finally there are some examples shown as test cases for the operational equivalence checker.

4.2.1 CHRat

CHRat is a paradigm for modular CHR called CHR with ask and tell. There is already the need for solvers for asks and tells for the built-in constraint system implementation. CHR does not provide entailment checks for CHR constraints, what is the reason why CHR constraints are not allowed to be used in a guard. With CHRat it is tried to internalize the requirement for asks and tells in the CHR solver itself. If the solved form of a constraint store containing $ask(c)$, where c is a constraint, contains the token $entailed(c)$, the constraint c is operationally entailed in this constraint store. [3]

A minimalist entailment solver can always be provided with the following simpagation rule:

$$c \setminus ask(c) \Leftrightarrow entailed(c)$$

And further rules of arbitarily complex entailment checks can be added to the entailment solver. [3]

Example 21. *CHRat Components for $leq/2$ and $min/3$ (this Example is taken from [3])*

The following CHR solver defines the constraint $leq/2$:

$$
\begin{array}{lrl}
\textbf{component}\ leq\_solver & & \\
\textbf{export}\ leq/2 & & \\
reflexive\ @ & leq(X,X) \Leftrightarrow & true. \\
antisymmetric\ @ & leq(X,Y),\ leq(Y,X) \Leftrightarrow & X = Y. \\
transitive\ @ & leq(X,Y),\ leq(Y,Z) \Rightarrow & leq(X,Z). \\
redundant\ @ & leq(X,Y) \setminus leq(X,Y) \Leftrightarrow & true.
\end{array}
$$

In order to use this constraint solver in CHRat, rules for an entailment solver are needed. The rule:

$$leq(X,Y) \backslash ask(leq(X,Y)) \Leftrightarrow entailed(leq(X,Y)).$$

is always assumed and provides a minimalist entailment solver, but in this example one more rule is needed:

$$ask(leq(X,X)) \Leftrightarrow entailed(leq(X,X)).$$

The following program uses the $leq/2$ constraints defined in the previously shown constraint solver.

$$
\begin{array}{lrl}
\textbf{component}\ min\_solver & & \\
\textbf{import}\ leq/2\ \textbf{from}\ leq\_solver & & \\
\textbf{export}\ min/3 & & \\
minLeft\ @ & min(X,Y,Z) \Leftrightarrow & leq(X,Y) \mid Z = X. \\
minRight\ @ & min(X,Y,Z) \Leftrightarrow & leq(Y,X) \mid Z = Y. \\
minGen\ @ & min(X,Y,Z) \Rightarrow & leq(Z,X), leq(Z,Y). \\
 & & \\
minAskLeft\ @ & ask(min(X,Y,Z)) \Leftrightarrow & leq(X,Y) \mid entailed(min(X,Y,X)). \\
minAskRight\ @ & ask(min(X,Y,Z)) \Leftrightarrow & leq(Y,X) \mid entailed(min(X,Y,Y)).
\end{array}
$$

Example 21 shows a program that uses CHR constraints in the guard. The interesting point now is, how one can transform such a CHRat program to a regular CHR program. For this the following definition is given in [3]:

Definition 22. Let $[\cdot]$: CHRat $\rightarrow$ CHR be defined for every CHRat rule by $[\cdot]$ as follows:

$$[rule \ @ \ H_k \setminus H_r \Leftrightarrow C_{built\text{-}in}, C_{CHR} \mid B.]$$
$$:= \begin{cases} rule\text{-}ask \ @ \ H_k, H_r \ \Rightarrow \ C_{built\text{-}in} \mid ask^*(C_{CHR}). \\ rule\text{-}fire \ @ \ H_k \setminus H_r, entailed^*(C_{CHR}) \ \Leftrightarrow \ C_{built\text{-}in} \mid B. \end{cases}$$

where H_k and H_r are the kept and the removed head constraints, $C_{built-in}$ and C_{CHR} are the built-in and the CHR constraints that are used in the guard and B is the body of the rule.

From this transformation rule, the transformations for simplification and propagation rules follow by immediate specialization. The image of a whole CHRat program (R, Σ) (where R is the set of CHRat rules and Σ is the signature of the set of constraint tokens) by $[\cdot]$ is the concatenation of images of the individual rules, with the propagation rules:

$$f(x_1, ..., x_k) \Rightarrow entailed(f(x_1, ..., x_k)).$$

implicitly added for each constraint declaration $(f/k) \ \in \ \Sigma$, if such a rule was not already written by the user in R.

Example 23. *Transformation of CHRat to CHR*

In this example the $min/3$ CHRat program shown in Example 21 is transformed to a regular CHR program according to definition 22.

$$\begin{array}{lrl}
rule\text{-}ask \ minLeft \ @ & min(X,Y,Z) \ \Rightarrow & ask_leq(X,Y). \\
rule\text{-}fire \ minLeft \ @ & min(X,Y,Z) \setminus entailed_leq(X,Y) \ \Leftrightarrow & Z = X. \\
rule\text{-}ask \ minRight \ @ & min(X,Y,Z) \ \Rightarrow & ask_leq(Y,X). \\
rule\text{-}fire \ minRight \ @ & min(X,Y,Z) \setminus entailed_leq(Y,X) \ \Leftrightarrow & Z = Y. \\
minGen \ @ & min(X,Y,Z) \Rightarrow & leq(Z,X), leq(Z,Y).
\end{array}$$

$$\begin{array}{lrl}
rule\text{-}ask \ minAskLeft \ @ & ask_min(X,Y,Z) \Rightarrow & ask_leq(X,Y). \\
rule\text{-}fire \ minAskLeft \ @ & ask_min(X,Y,Z) \setminus entailed_leq(X,Y) \ \Leftrightarrow & entailed_min(X,Y,X). \\
rule\text{-}ask \ minAskRight \ @ & ask_min(X,Y,Z) \Rightarrow & ask_leq(Y,X). \\
rule\text{-}fire \ minAskRight \ @ & ask_min(X,Y,Z) \setminus entailed_leq(Y,X) \ \Leftrightarrow & entailed_min(X,Y,Y).
\end{array}$$

4.2.2 Replacing Guards by 'ask' and 'entailed' Constraints

To replace a rule with a built-in in a guard with equivalent rules that use a constraint solver to replace that built-in a constraint solver that has the constraint theory representing the built-in

implemented is required. Additionally a transformation rule that can be applied to any CHR rule with built-ins in its guard is needed. In the following it is assumed that appropriate constraint solvers are present so that focus can be put on the actual transformation process.

Example 24. *Rule identifiers*

Program P:

$$\begin{aligned} &rule\text{-}1\ @ && test(A,B)\ \Rightarrow && A \leq B | result(A).\\ &rule\text{-}2\ @ && test(A,B)\ \Rightarrow && A \leq B | result(B). \end{aligned}$$

Let a constraint solver for less or equal be present, then the following program would be the transformed program of P with a similar transformation like used in Example 23

$$\begin{aligned} &rule\text{-}ask\text{-}1\ @ && test(A,B)\ \Rightarrow && ask\_leq(A,B).\\ &rule\text{-}fire\text{-}1\ @ && test(A,B) \setminus entailed\_leq(A,B)\ \Leftrightarrow && result(A).\\ &rule\text{-}ask\text{-}2\ @ && test(A,B)\ \Rightarrow && ask\_leq(A,B).\\ &rule\text{-}fire\text{-}2\ @ && test(A,B) \setminus entailed\_leq(A,B)\ \Leftrightarrow && result(B). \end{aligned}$$

If program P and the transformed program are called with the query:

$$leq(A,B), test(A,B)$$

the final state of program P would always be $leq(A,B), test(A,B), result(A), result(B)$, the transformed program could also reach this final state, but it could also fire rule 1 or 2 twice instead of each once. This shows that it is important to have an identifier for each rule that needs to be added to the ask and entailed constraints, so that only the rule that asked for the entailment check can fire with the resulting entailed constraint.

The transformed program with an added identifier for the rules would look like this:

$$\begin{aligned} &rule\text{-}ask\text{-}1\ @ && test(A,B)\ \Rightarrow && ask\_leq(A,B,1).\\ &rule\text{-}fire\text{-}1\ @ && test(A,B) \setminus entailed\_leq(A,B,1)\ \Leftrightarrow && result(A).\\ &rule\text{-}ask\text{-}2\ @ && test(A,B)\ \Rightarrow && ask\_leq(A,B,2).\\ &rule\text{-}fire\text{-}2\ @ && test(A,B) \setminus entailed\_leq(A,B,2)\ \Leftrightarrow && result(B). \end{aligned}$$

Now the transformed program behaves like the original program P.

Example 25. *Order of constraints and variables*

Program P:

$$rule\text{-}1\ @ \qquad test(A,B), c(X), c(Y)\ \Rightarrow \qquad A \leq B | result(X).$$

After a similar transformation as in example 24 the transformed program looks like:

$$\begin{aligned} &rule\text{-}ask{-}1 \ @ && test(A,B), c(X), c(Y) \ \Rightarrow && ask\_leq(A,B,1). \\ &rule\text{-}fire{-}1 \ @ && test(A,B), c(X), c(Y) \setminus entailed\_leq(A,B,1) \ \Leftrightarrow && result(X). \end{aligned}$$

If program P and the transformed program are called with the query:

$$leq(A,B), test(A,B), c(1), c(2)$$

the final state of program P would always be $leq(A,B), test(A,B), result(1), result(2)$, but the transformed program could use the same constellation of $c(1), c(2)$ twice since it gets two different entailed constraints.

To prevent this the order of the constraints with their variables needs to be stored in the ask and entailed constraints.

The transformed program with an added store for the constraints with their variables would look like this:

$$\begin{aligned} &rule\text{-}ask \ 1 \ @ \\ &\quad test(A,B), c(X), c(Y) && \Rightarrow ask\_leq(A,B,1,t(test(A,B), c(X), c(Y))). \\ &rule\text{-}fire \ 1 \ @ \\ &\quad test(A,B), c(X), c(Y) \setminus entailed\_leq(A,B,1,T) && \Leftrightarrow T == t(test(A,B), c(X), c(Y)) \mid result(X). \end{aligned}$$

Now the transformed program behaves like the original program P.

Note: A query like $leq(A,B), test(A,B), c(1), c(1)$ might still cause one constraint constellation to fire twice while the other never fires but that has no effect on the result since it cannot fire more often than it is supposed to and both constellations are equivalent.

The thoughts from example 24 and example 25 lead to the following definition:

Definition 26. Iff a constraint solver that can represent a built-in b_x is present in a program P, then let $[\cdot] : CHR_{guardbuilt-in} \rightarrow CHR_{constraint}$ be defined for every CHR rule of the program P by $[\cdot]$ as follows:

$$[rule \ i \ @ \ H_k \setminus H_r \Leftrightarrow C_{built\text{-}ins}, b_1(x_{1,1}, ...x_{1,k}), ..., b_n(x_{n,1}, ..., x_{n,l}) \mid B.]$$

$$:= \begin{cases} rule \ i \ ask \ @ \\ H_k, H_r \ \Rightarrow \ C_{built\text{-}ins} \mid ask\_b_1(x_{1,1}, ...x_{1,k}, i, t(H_k, H_r)), ..., ask\_b_n(x_{n,1}, ..., x_{n,l}, i, t(H_k, H_r)). \\ rule \ i \ fire \ @ \\ H_k \setminus H_r, entailed\_b_1(x_{1,1}, ...x_{1,k}, i, T), ..., entailed\_b_n(x_{n,1}, ..., x_{n,l}, i, T) \ \Leftrightarrow \ C_{built\text{-}ins}, T == t(H_k, H_r) \mid B. \end{cases}$$

where H_k and H_r are the kept and the removed head constraints, $C_{built\text{-}in}$ are the built-in constraints that are used in the guard and will remain in the guard, $b_1(x_{1,1}, ...x_{1,k}), ..., b_n(x_{n,1}, ..., x_{n,l})$ are the built-in constraints from the guard that have appropriate constraint solvers and are supposed to be replaced with CHR constraints, B is the body and i is an unique identifier of the rule.

The entailment solver needs to carry the rule identifier i and the term $t(H_k, H_r)$ over to the entailed constraints, they are needed to identify which rule asked for the entailment check and to ensure that the right variables are used.

4.2.3 Modified CHR Parser

To add support for more built-ins to the operational equivalence checker the transformation described in definition 26 needs to be executed on the tested CHR program for every rule that has those built-ins in its guard. It is convenient to modifie the `chrparser.pl` in a way that lets it generate the transformed program, since all the necessary information is present during the parsing process. This modified version of the `chrparser.pl` is called `chrparserOPEQ.pl`.

As mentioned in section 4.1.2 the operational equivalence checker is able to handle propagation rules since it does not need to add a propagation history for its queries. So the first modification to the parser is to add support for propagation rules. To do this a additional case for $parse\_actrule/7$ is added that tries to match a line from the program on $(LHS ==> RHS)$ and parses it in a similar way the original parser did for simpagation and simplification rules.

As new supported built-ins $B_{new} = \{\leq , \geq , < , > , ==\}$ were chosen. For adding the transformation first the $read\_rules/3$ needed to be modified and a parameter was added:

$$read\_rules(File, NewFile, Rules, CHRC)$$

$File$ and $CHRC$ are unchanged. $NewFile$ is a String with the path to the file where the transformed program will be saved. $Rules$ is almost unchanged, the only difference is, that the built-ins from B_{new} were removed from the guard and appropriate constraints were added to the kept head. This is not the cleanest way to extract this information, but it makes checks for operational equivalence easier since there is no need for big changes at the existing code. The predicate itself calls some predicates that write header information to the $NewFile$ (necessary code like the constraint definitions and the constraint solvers for the built-ins from B_{new} that will be shown in the following sections). It also calls the predicate $extract\_rules/4$ where the actual program transformation is prepared.

$$extract\_rules(Clauses, Stream, Rules, I)$$

$Clauses$ is a list containing the clauses from the program that is parsed and transformed, $Stream$ is the stream to the file where the transformd program is written to and I is the identifier for each

rule. Initially I is set to zero and with each added rule that is added to the transformed program it is incremented.

The actual transformation is performed by the following predicate:

$$parse\_actrule(I, Stream, Term, KH, RH, G, B)$$

$Term$ is a term of the from $(LHS\ ==>\ RHS)$ or $(LHS\ <=>\ RHS)$ where the rule name has been removed. KH and RH are lists of CHR constraints represented as Prolog terms and represent the kept and the removed head. As mentioned earlier KH also contains terms representing the constraints of B_{new} if any of those were present in the guard of the original rule. Any $\geq, >$ that were present have been replaced by $\leq, <$ with flipped arguments, so that no constraint solvers for $\geq, >$ are needed. G is a list representing the guard, all built-ins that are in B_{new} have been removed. B is a list with the constraints of the body of the rule.

The predicate $write\_ask/4$ writes the 'rule i ask' propagation rule according to definition 26 if it is needed and the predicate $write\_rule/5$ writes the rule to the file for the transformed program. If any built-ins from B_{new} were present in this rule this looks like the 'rule i fire' from definition 26, otherwise it just writes the rule like it appears in the original program.

This gives leads to an executable CHR program, that should behave very similar to the original program. A difference to the original program is, that some constraints containing information for the constraint solvers will be present in the final states. The final states might also contain some leftover ask and $entailed$ constraints. For those a cleanup needs to be done (see section 4.2.5).

4.2.4 Adding Constraint Solvers

To make the rules that have been transformed according to definition 26 work, constraint solvers for the replaced built-ins are needed. For those the information needs to be represented as CHR constraints and they need to have an entailment solver. This section describes the constraint solvers that the chrparserOPEQ.pl adds to the header of a transformed program.

Adding a failed state

As explained in section 2.2.2 failed built-ins should lead to a failed state and all failed states should be considered equivalent. The buil-in $fail$ that Prolog provides is not useful to represent failed states, since this would cause the consult to fail what would cause the complete test to fail. To check for failed states the CHR constraint $builtin\_fail/0$ is defined. If a constraint solver finds any conflicting constraints that represent information for the built-ins it will add a $builtin\_fail/0$ to the constraint store. Additional rules of the form:

$$builtin\_fail \setminus C \ \Leftrightarrow\ true.$$

For easier understanding C can be considered a wildcard, if $builtin\_fail/0$ shows up at any time during the computation, it will remove all other constraints until only one $builtin\_fail/0$ remains as final state. Since there are no wild cards for constraints in CHR, a extra rule for each defined constraint is be added to the transformed program right after the definition of the constraints.

Note: Failed states represented with the CHR constraint $builtin\_fail$ are not necessarily equivalent states, since the global variables can be different. Since failed states should be considered equivalent an additional check is needed when checking for state equivalence.

Adding a Constraint Solver for less or equal

To represent the built-in $\leq$ a CHR constraint $leq/2$ is chosen. The basic constraint solver is similar to the one in example 21 that was taken from [4]:

$$
\begin{array}{lrl}
redundant\ @ & leq(X,Y) \setminus leq(X,Y) \Leftrightarrow & true. \\
reflexive\ @ & leq(X,X) \Leftrightarrow & true. \\
antisymmetric\ @ & leq(X,Y),\ leq(Y,X) \Leftrightarrow & X = Y. \\
transitive\ @ & leq(X,Y),\ leq(Y,Z) \Rightarrow & leq(X,Z).
\end{array}
$$

Also rules for the entailment solver need to be added

$$
\begin{array}{ll}
leq(X,Y) \setminus ask\_leq(X,Y,I,T) & \Leftrightarrow entailed\_leq(X,Y,I,T). \\
ask\_leq(X,X,I,T) & \Leftrightarrow entailed\_leq(X,X,I,T).
\end{array}
$$

For the case that on or both arguments in an $ask\_leq/4$ constraint are bound variables the following entailment rules are added to let the behavior of the constraint solver be closer to that of the built-in.

$$
\begin{array}{ll}
ask\_leq(X,Y,I,T) & \Leftrightarrow number(X), number(Y), X \leq Y \mid entailed\_leq(X,Y,I,T). \\
leq(X,Z) \setminus ask\_leq(X,Y,I,T) & \Leftrightarrow number(Z), number(Y), Z \leq Y \mid entailed\_leq(X,Y,I,T). \\
leq(Y,X) \setminus ask\_leq(Z,X,I,T) & \Leftrightarrow number(Z), number(Y), Z \leq Y \mid entailed\_leq(Z,X,I,T).
\end{array}
$$

The first of those rules is for the case that the original built-in can actually be used. The other two rules are for the case that a bound variable is compared with a unbound variable where information on the unbound variable is present.

Adding a Constraint Solver for less

To represent the built-in $<$ a CHR constraint $less/2$ is chosen. The constraint solver is analogical to the constraint solver for $\leq$.

$$
\begin{array}{lrl}
redundant\ @ & less(X,Y) \setminus less(X,Y) \Leftrightarrow & true. \\
reflexive\ @ & less(X,X) \Leftrightarrow & builtin\_fail. \\
antisymmetric\ @ & less(X,Y),\ less(Y,X) \Leftrightarrow & builtin\_fail. \\
transitive\ @ & less(X,Y),\ less(Y,Z) \Rightarrow & less(X,Z). \\
 & less(X,Y) \setminus leq(X,Y) \Leftrightarrow & true. \\
 & leq(Y,X), less(X,Y) \Leftrightarrow & builtin\_fail.
\end{array}
$$

There is a rule that removes redundant $leq/2$ constraints to keep the stored information minimalistic. Since a $<$ also gives information for $\leq$ the entailment solver needs to handle those as well.

$$
\begin{array}{ll}
less(X,Y) \setminus ask\_leq(X,Y,I,T) & \Leftrightarrow entailed\_leq(X,Y,I,T). \\
less(X,Y) \setminus ask\_less(X,Y,I,T) & \Leftrightarrow entailed\_less(X,Y,I,T). \\
less(X,Z) \setminus ask\_leq(X,Y,I,T) & \Leftrightarrow number(Z), number(Y), Z \leq Y \mid entailed\_leq(X,Y,I,T). \\
less(Y,X) \setminus ask\_leq(Z,X,I,T) & \Leftrightarrow number(Z), number(Y), Z \leq Y \mid entailed\_leq(Z,X,I,T). \\
ask\_less(X,Y,I,T) & \Leftrightarrow number(X), number(Y), X < Y \mid entailed\_less(X,Y,I,T). \\
less(X,Z) \setminus ask\_less(X,Y,I,T) & \Leftrightarrow number(Z), number(Y), Z < Y \mid entailed\_less(X,Y,I,T). \\
less(Y,X) \setminus ask\_less(Z,X,I,T) & \Leftrightarrow number(Z), number(Y), Z < Y \mid entailed\_less(Z,X,I,T).
\end{array}
$$

Adding a Constraint Solver for equal

To represent the built-in $==$ a CHR constraint $eq/2$ is chosen.

$$
\begin{array}{lrl}
redundant\ @ & eq(X,Y) \setminus eq(X,Y) \Leftrightarrow & true. \\
reflexive\ @ & eq(X,X) \Leftrightarrow & true. \\
symmetric\ @ & eq(X,Y) \Rightarrow & eq(Y,X). \\
transitive\ @ & eq(X,Y),\ eq(Y,Z) \Rightarrow & eq(X,Z).
\end{array}
$$

$$
\begin{aligned}
eq(X,Y) \backslash leq(X,Y) &\Leftrightarrow && true.\\
eq(X,Y), less(X,Y) &\Leftrightarrow && builtin\_fail.\\
eq(X,Y), less(Y,X) &\Leftrightarrow && builtin\_fail.\\
eq(X,Y) &\Leftrightarrow && nonvar(X), nonvar(Y), X \backslash == Y \mid builtin\_fail.\\
eq(X,Y), less(X,Z) &\Rightarrow && less(Y,Z).\\
eq(X,Y), less(Z,X) &\Rightarrow && less(Z,Y).\\
eq(X,Y), leq(X,Z) &\Rightarrow && leq(Y,Z).\\
eq(X,Y), leq(Z,X) &\Rightarrow && leq(Z,Y).
\end{aligned}
$$

There is a rule that removes redundant $leq/2$ constraints to keep the stored information minimalistic. Since a $==$ also gives information for $\leq$ the entailment solver needs to handle those as well.

$$
\begin{aligned}
eq(X,Y) \backslash ask\_eq(X,Y,I,T) &\Leftrightarrow entailed\_eq(X,Y,I,T).\\
eq(Y,X) \backslash ask\_eq(X,Y,I,T) &\Leftrightarrow entailed\_eq(X,Y,I,T).\\
ask\_eq(X,X,I,T) &\Leftrightarrow entailed\_eq(X,X,I,T).\\
ask\_eq(X,Y,I,T) &\Leftrightarrow X == Y \mid entailed\_eq(X,Y,I,T).\\
eq(X,Z) \backslash ask\_eq(X,Y,I,T) &\Leftrightarrow Z == Y \mid entailed\_eq(X,Y,I,T).\\
eq(Y,X) \backslash ask\_eq(Z,X,I,T) &\Leftrightarrow Z == Y \mid entailed\_eq(Z,X,I,T).\\
eq(X,Z) \backslash ask\_leq(X,Y,I,T) &\Leftrightarrow number(Z), number(Y), Z \leq Y \mid entailed\_leq(X,Y,I,T).\\
eq(Y,X) \backslash ask\_leq(Z,X,I,T) &\Leftrightarrow number(Z), number(Y), Z \leq Y \mid entailed\_leq(Z,X,I,T).
\end{aligned}
$$

4.2.5 Cleanup

As mentioned earlier, the resulting final states of the transformed code can have constraints in it that the original would not have. These are the constraints storing information for the constraint solver that represents the built-ins from B_{new} and leftover ask and entailed constraints. The constraints storing information for the constraint solvers need to be present in both states in order for them to be equivalent. Ask and entailed constraints are only relevant during the computation to check if a rule can fire with certain constraints (see Example 27). This means that during the check for operational equivalence, those leftover constraints need to be removed from any final states before those are checked for equivalence. For this purpose the module `cleanup.pl` offers a predicate $cleanup/2$ that removes any leftover ask or entailed constraints.

Example 27. *Consider the following rule:*

$$c(A,B) \;\Leftrightarrow\; A < B, B < A \mid true.$$

Since this rule can never fire, it can be added to any program that has the constraint $c/2$ defined without changing its functionality. If the transformed version of a program with this rule is consulted with the following query:

$$less(A, B), c(A, B).$$

the resulting final state would contain an $ask\_less(A, B, i)$ and an $entailed\_less(A, B, i, t(c(A, B)))$. These constraints are leftovers from a check if this rule can fire and obviously need to be removed from the final state, since otherwise a rule that did not fire would have influenced on the final state.

4.2.6 Limitations

In comparison to the limitations mentioned in section 4.1.2 the operational equivalence checker now supports the built-ins from B_{new} and works with propagation rules. Constraints with the names $leq/2, ask\_leq/4, entailed\_leq/4, less/2, ask\_less/4, entailed\_less/4, eq/2, ask\_eq/4,$ $entailed\_eq/4, builtin\_fail/0$ are not allowed to be used in the tested programs, since they are defined by the constraint solvers in the transformed code. As a workaround for this limitation these constraints can easily be renamed in a tested program. The rest of the limitations have not changed.

4.3 Test cases and examples

The examples in this section are all tested in the same way. The module `opeq.pl` is consulted in SWI-Prolog and the predicate $opeq/2$ is called with the file-paths of both files.

Example 28. *Test of Example 20*

Program 1:

```
1 :- use_module(library(chr)).
2 :- chr_constraint a/0,b/0,helloworld/0.
3
4 a <=> helloworld.
5 b <=> helloworld.
```

Program 2:

```
1 :- use_module(library(chr)).
2 :- chr_constraint a/0,b/0,helloworld/0.
3
4 a <=> b.
5 b <=> helloworld.
```

Result:

```
The programs are operationally equivalent.
```

Example 29. *Test of Example 19*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint min/1.

min(X),min(Y) <=> X<Y | min(X).
min(X),min(Y) <=> X==Y | min(X).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint min/1.

min(X),min(Y) <=> X=<Y | min(X).
```

Result:

```
==========================================================================
For the minimal State:
minState([leq(A,B),min(A),min(B)],[],[A,B])

file 1 results in:
[leq(C,D),min(D),min(C),globs([C,D])]

while file 2 results in:
[leq(E,F),min(E),globs([E,F])]

these states are not joinable
==========================================================================


The programs are not operationally equivalent,
1 minimal state(s) are not joinable.
```

As expected, the minimal states containing the equal and the less constraint don't cause any problems, but the minimal state with less or equal can not fire any rules in program 1.

Example 30. *Test of splitting a propagation rule in several propagation rules, switching the position of constraints in the rules and replacing the built-in $<$ with the built-in $>$ and the built-in $\leq$ with the built-in $\geq$ by flipping the arguments.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test1/2, test2/2, test3/2, test/1.

test1(A,B),test2(C,D),test3(B,D) ==>
        A=<B , C<D, B==D | test(A), test(C) , test(B).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test1/2, test2/2, test3/2, test/1.

test1(A,B),test2(C,D),test3(B,D) ==> A=<B , C<D , B==D | test(A).
test3(B,D),test1(A,B),test2(C,D) ==> A=<B , B==D , C<D | test(C).
test2(C,D),test1(A,B),test3(B,D) ==> B>=A , D>C , B==D | test(B).
```

Result:

```
The programs are operationally equivalent.
```

Example 31. *Test of two programs with similar looking final states, but with different variables.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test1/2, test2/2, test3/2, test/1.

test1(A,B) \ test2(C,D) <=> A=<B | test(C).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test1/2, test2/2, test3/2, test/1.

test1(A,B) \ test2(C,D) <=> A=<B | test(D).
```

Result:

```
======================================================================
For the minimal State:
minState([test1(A,B),leq(A,B),test2(C,D)],[],[A,B,C,D])

file 1 results in:
[leq(E,F),test1(E,F),test(G),globs([E,F,G,H])]

```

```
while file 2 results in:
[leq(I,J),test1(I,J),test(K),globs([I,J,L,K])]

these states are not joinable
=====================================================================
=====================================================================
For the minimal State:
minState([test1(A,B),leq(A,B),test2(C,D)],[],[A,B,C,D])

file 1 results in:
[leq(E,F),test1(E,F),test(G),globs([E,F,G,H])]

while file 2 results in:
[leq(I,J),test1(I,J),test(K),globs([I,J,L,K])]

these states are not joinable
=====================================================================


The programs are not operationally equivalent,
2 minimal state(s) are not joinable.
```

The state equivalence checker noticed, that the variables are different.

Example 32. *Test of a modified version of Example 31, where the different variables are unified in the guard.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test1/2, test2/2, test3/2, test/1.

test1(A,B) \ test2(C,D) <=> A=<B, C=D | test(C).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test1/2, test2/2, test3/2, test/1.

test1(A,B) \ test2(C,D) <=> A=<B, C=D| test(D).
```

Result:

```
The programs are operationally equivalent.
```

Since the previously different variables have now been unified with each other, this moddified version is operational equivalent.

Example 33. *Test where a rule that cannot fire is added to a program*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(A,B) <=> A<B | test(A).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(A,B) <=> A==B, A<B | test(B).
test(A,B) <=> A<B | test(A).
```

Result:

```
The programs are operationally equivalent.
```

Since the added rule cannot fire and its minimal state leads to a failed final state (as described in section 4.2.4) the two programs are operational equivalent.

Example 34. *Test where the user defined CHR constraints are different.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test/3, test/1.
```

Result:

```
The two programs have different CHR constraints defined.
```

Since there are different constraints defined no further tests are done.

Example 35. *Test with a redundant rule where the constraint solver needs to compare variables and numbers.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(A,B) <=> A<5 | test(B).
test(A,B) <=> A<6 | test(B).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(A,B) <=> A<6 | test(B).
```

Result:

```
The programs are operationally equivalent.
```

When the minimal state of the first rule of program 1 is checked, the constraint-solver for less in program 2 is asked if $A < 6$ is true and it only has the information that $A < 5$ is true. Since the constraint solvers that are used for the built-ins B_new can to some degree work with numbers this example is working.

Example 36. *Test with two obviously not equivalent programs.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(A,B) <=> A<B | test(B).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(A,B) <=> A==B | test(B).
```

Result:

```
======================================================================
For the minimal State:
minState([less(A,B),test(A,B)],[],[A,B])

file 1 results in:
```

```
[less(C,D),test(D),globs([C,D])]

while file 2 results in:
[less(E,F),test(E,F),globs([E,F])]

these states are not joinable
========================================================================
========================================================================
For the minimal State:
minState([eq(A,B),test(A,B)],[],[A,B])

file 1 results in:
[eq(C,D),eq(D,C),test(D,C),globs([D,C])]

while file 2 results in:
[eq(E,F),eq(F,E),test(E),globs([F,E])]

these states are not joinable
========================================================================


The programs are not operationally equivalent,
2 minimal state(s) are not joinable.
```

Example 37. *Test with two logically equivalent programs to determine the maximum. (This example is taken from [4, p. 129])*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint max/3.

max(X,Y,Z) <=> X=<Y | Z=Y.
max(X,Y,Z) <=> Y<X | Z=X.
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint max/3.

max(X,Y,Z) <=> X<Y | Z=Y.
max(X,Y,Z) <=> Y=<X | Z=X.
```

Result:

```
==========================================================================
For the minimal State:
minState([leq(A,B),max(A,B,C)],[],[A,B,C])

file 1 results in:
[leq(D,E),globs([D,E,E])]

while file 2 results in:
[leq(F,G),max(F,G,H),globs([F,G,H])]

these states are not joinable
==========================================================================
==========================================================================
For the minimal State:
minState([leq(A,B),max(B,A,C)],[],[A,B,C])

file 1 results in:
[leq(D,E),max(E,D,F),globs([D,E,F])]

while file 2 results in:
[leq(G,H),globs([G,H,H])]

these states are not joinable
==========================================================================


The programs are not operationally equivalent,
2 minimal state(s) are not joinable.
```

Since the minimal states with less or equal lead to different final states, those two programs are not operationally equivalent.

5 Adding support for more Built-ins to the Confluence Checker

This section describes how the support for the built-ins from $B_{new} = \{<, >, \leq, \geq, ==\}$ has been added to the confluence checker. The same code transformations as described in section 4.2.2 are used to realize this.

5.1 Problem with the Transformed Code

The easiest way to add support to those built-ins would be if a program could be checked for confluence by simply calling the confluence checker with the transformed code. Unfortunately there are two points that prevent this from being this easy. The first problem is that the constraint solvers for the added built-ins use built-ins that the confluence checker does not support. The second problem is, that again there might be leftover ask or entailed constraints that need to be removed before the final states can be checked for equivalence. (Just like in section 4.2.5) This means that the confluence checker needs to check for critical pairs in the original code where some handling for the unsupported built-ins needs to be added and that after a critical pair was checked a cleanup needs to be done to remove ask and entailed constraints.

5.2 Necessary modifications

The modules that need to be modified are the `confcheck.pl` and the `criticalpairs.pl`.

The first step is to let the $check\_confluence/1$ predicate from the `confcheck.pl` call the $read\_rules/4$ predicate from the `chrparserOPEQ.pl` to obtain a file with the transformed code that is needed later to run the tests.

The critical pairs are created just like before, with one modification to the $critical\_pair/5$ predicate from the `criticalpairs.pl` module. The built-ins from B_{new} are now filtered from the guards of a potential critical pair and constraints representing their information are added to the kept head.

The last modification is done to the $states_joinable/3$ predicate from the `confcheck.pl` module. This predicate is now called with the transformed program and therefore the $cleanup/2$ predicate from the `cleanup.pl` module is called before the final states are checked for equivalence to

get rid of any leftover ask or entailed constraints. Additionally to the normal state equivalence check those final states are also checked for the $builtin_fail$ constraint, and if it is present in both final states they are considered equivalent even if the state equivalence check failed.

5.3 Limitations

Two limitations have changed in comparison to section 3.4. One is that the built-ins from B_{new} are now supported. The other is that constraints with the names $leq/2, ask\_leq/4, entailed\_leq/4, less/2, ask\_less/4, entailed\_less/4, eq/2, ask\_eq/4, entailed\_eq/4, builtin\_fail/0$ are not allowed to be used in the tested programs, since they are defined by the constraint solvers in the transformed code. As a workaround for this limitation these constraints can easily be renamed in a tested program. There is still no syntax check performed and no support for propagation rules.

Notre: When creating critical pairs built-ins from B_{new} are ignored. This means that there are some trivial critical pairs with failed built-ins checked for confluence. Those critical pairs have no influence on the actual result, since due to the rules for failed states they will result in final states with just one $builtin\_fail/0$ constraint (see section 4.2.4.

5.4 Testcases and examples

The examples in this section are all tested in the same way. The module `confcheck.pl` is consulted in SWI-Prolog and the predicate $check\_confluence/2$ is called with the file-path of the file. First some examples that came with the confluence checker [1] are tested and after that some examples using built-ins from B_{new} are tested.

Example 38. *Very simple program with only trivial overlaps [1]*

```
1 :- use_module(library(chr)).
2 :- chr_constraint p/0, q/0.
3
4 p <=> true.
5 q <=> true.
```

Result:

```
1 Checking confluence of CHR program in ~\simple1.pl...
2
3 The CHR program in ~\simple1.pl is confluent.
```

Example 39. *Program with one not joinable overlap [1]*

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

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

Result:

```
Checking confluence of CHR program in ~\simple2.pl...

========================================================================
The following critical pair is not joinable:
state([q],[],[])
state([],[false],[])

This critical pair stems from the critical ancestor state:
[p]

with the overlapping part:
[ (p,p)]

of the following two rules:
p<=>q
p<=>false
========================================================================

The CHR program in ~\simple2.pl is NOT confluent!
1 non-joinable critical pair(s) found!
```

Example 40. *Program with a non trivial critical pair and some trivial critical pairs [1]*

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

p,q <=> true.
q,r <=> true.
```

Result:

```
Checking confluence of CHR program in ~\simple3.pl...

========================================================================
```

```
The following critical pair is not joinable:
state([r],[true],[])
state([p],[true],[])

This critical pair stems from the critical ancestor state:
[p,q,r]

with the overlapping part:
[ (q,q)]

of the following two rules:
p,q<=>true
q,r<=>true
================================================================================

The CHR program in ~\simple3.pl is NOT confluent!
1 non-joinable critical pair(s) found!
```

Example 41. *Program with guards that has some overlaps with an inconsistent built-in store and some critical pairs [1]*

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

p(X) <=> X = 1 | true.
p(X) <=> X = 2 | true.
p(2) <=> true.
```

Result:

```
Checking confluence of CHR program in ~\simple4.pl...

The CHR program in ~\simple4.pl is confluent.
```

Example 42. *A constraint solver for xor [1]*

```
:- use_module(library(chr)).
:- chr_constraint xor/1.

xor(X), xor(X) <=> xor(0).
xor(1) \ xor(0) <=> true.
```

Result:

```
Checking confluence of CHR program in ~\xor.pl...

The CHR program in ~\xor.pl is confluent.
```

Example 43. *A program with built-ins from B_{new} in the guards that has overlaps with inconsistent built-ins that during the confluence check are represented by CHR constraints due to the program transformation*

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(X,Y) <=> X==Y | test(yay).
test(X,Y)  <=> X<Y | test(doh).
test(X,Y)  <=> X>Y | test(oh).
```

Result:

```
Checking confluence of CHR program in ~\example6c.pl...

The CHR program in ~\example6c.pl is confluent.
```

Example 44. *A modified version of Example 43 where the overlaps now lead to non joinable critical pairs*

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(X,Y) <=> X==Y | test(yay).
test(X,Y)  <=> X=<Y | test(doh).
test(X,Y)  <=> X>=Y | test(oh).
```

Result:

```
Checking confluence of CHR program in ~\example7c.pl...

========================================================================
The following critical pair is not joinable:
state([eq(A,B),leq(C,D),test(yay)],[A=C,B=D],[A,B,C,D])
state([eq(A,B),leq(C,D),test(doh)],[A=C,B=D],[A,B,C,D])

This critical pair stems from the critical ancestor state:
[test(A,B),A=C,B=D,A==B,C=<D]

```

```
with the overlapping part:
[ (test(A,B),test(C,D))]

of the following two rules:
test(E,F)<=>E==F|test(yay)
test(K,L)<=>K=<L|test(doh)
================================================================================

================================================================================
The following critical pair is not joinable:
state([eq(A,B),leq(C,D),test(yay)],[A=D,B=C],[A,B,D,C])
state([eq(A,B),leq(C,D),test(oh)],[A=D,B=C],[A,B,D,C])

This critical pair stems from the critical ancestor state:
[test(A,B),A=D,B=C,A==B,D>=C]

with the overlapping part:
[ (test(A,B),test(D,C))]

of the following two rules:
test(E,F)<=>E==F|test(yay)
test(K,L)<=>K>=L|test(oh)
================================================================================

================================================================================
The following critical pair is not joinable:
state([leq(A,B),leq(C,D),test(doh)],[A=D,B=C],[A,B,D,C])
state([leq(A,B),leq(C,D),test(oh)],[A=D,B=C],[A,B,D,C])

This critical pair stems from the critical ancestor state:
[test(A,B),A=D,B=C,A=<B,D>=C]

with the overlapping part:
[ (test(A,B),test(D,C))]

of the following two rules:
test(E,F)<=>E=<F|test(doh)
test(K,L)<=>K>=L|test(oh)
================================================================================

```

```
51 The CHR program in ~\example7c.pl is NOT confluent!
52 3 non-joinable critical pair(s) found!
```

6 Adding a Confluence Check to the Operational Equivalence Checker

The goal is to create a program that first checks two programs for confluence and performs an operational equivalence check. This means that the confluence checker should test both programs in different modules (just like in section 4.1.1). The program should terminate when at least one of the input programs is not confluent without starting the operational equivalence test.

6.1 Necessary modifications

The module opeq.pl now has the predicate

$$confopeq(File1, File2)$$

which is very similar to the $opeq/2$ that was described in section 4. The major difference is, that before it checks for operational equivalence, it calls the following predicate with both programs that are tested

$$check\_confluence1(File, FileExe, C, N)$$

where $File$ is the path to the original file, $FileExe$ is the file-path to the transformed program, C is either 1 or 2 to represent if it is the first or the second program and N is the number of non joinable critical pairs. The predicate works like described in section 5.2 just that it now returns the number of non joinable critical pairs and that the $states\_joinable/3$ has been modified to a $states\_joinable/4$ to accept the parameter C so that it either consults the file in the `consult1.pl` or the `consult2.pl` module.

After the confluence checks both N are added up. If the result is greater than zero at least one program is not confluent and the checker terminates, otherwise the operational equivalence check is continued just like it is described in section 4.

6.2 Limitations

Since the confluence checker has not been modified in its essence since section 5.2, this program has all the limitations described in section 5.3. Compared to the limitations mentioned

in section 4.2.6 the tested programs no longer need to be well-behaved, they just need to be terminating. But the support for propagation rules is lost.

Since this is the last modification, the limitations are summed up once more:

- no support for propagation rules
- only the built-ins $true$, $=/2$, $\leq /2$, $\geq /2$, $<$, $>$ and $==$ are supported
- the tested programs need to terminate
- only valid CHR programs should be checked
- Constraints with the names $leq/2, ask\_leq/4, entailed\_leq/4, less/2, ask\_less/4, entailed\_less/4, eq/2, ask\_eq/4, entailed\_eq/4, builtin\_fail/0$ are not allowed to be used in the tested programs

6.3 Testcases and examples

The examples in this section are all tested in the same way. The module `opeq.pl` is consulted in SWI-Prolog and the predicate $confopeq/2$ is called with the file-paths of the files that are tested.

Example 45. *Test of Example 43 and a slightly modified version that should be operational equivalent.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(X,Y) <=> X==Y | test(yay).
test(X,Y)  <=> X<Y | test(doh).
test(X,Y)  <=> X>Y | test(oh).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(X,Y)  <=> Y<X | test(oh).
test(X,Y) <=> X==Y | test(yay).
test(X,Y)  <=> Y>X | test(doh).
```

Result:

```
Checking confluence of CHR program in ~\example6c.pl...

```

```
The CHR program in E:\uni\bachelor\opeq-examples\example6c.pl is confluent.


Checking confluence of CHR program in ~\example6c-2.pl...

The CHR program in ~\example6c-2.pl is confluent.



The programs are operationally equivalent.
```

Example 46. *Test of Example 44 and a slightly modified version that should be operational equivalent, but does not matter since neither program is confluent.*

Program 1:

```
:- use_module(library(chr)).
:- chr_constraint test/2, test/1.

test(X,Y) <=> X==Y | test(yay).
test(X,Y)  <=> X=<Y | test(doh).
test(X,Y)  <=> X>=Y | test(oh).
```

Program 2:

```
:- use_module(library(chr)).
:- chr_constraint test/1, test/2.

test(X,Y)  <=> Y>=X | test(doh).
test(X,Y) <=> Y==X | test(yay).
test(X,Y)  <=> Y==X | test(oh).
```

Result:

```
Checking confluence of CHR program in ~\example7c.pl...

========================================================================
The following critical pair is not joinable:
state([eq(A,B),leq(C,D),test(yay)],[A=C,B=D],[A,B,C,D])
state([eq(A,B),leq(C,D),test(doh)],[A=C,B=D],[A,B,C,D])

This critical pair stems from the critical ancestor state:
[test(A,B),A=C,B=D,A==B,C=<D]
```

```
10
11 with the overlapping part:
12 [ (test(A,B),test(C,D))]
13
14 of the following two rules:
15 test(E,F)<=>E==F|test(yay)
16 test(K,L)<=>K=<L|test(doh)
17 ==========================================================================
18
19 ==========================================================================
20 The following critical pair is not joinable:
21 state([eq(A,B),leq(C,D),test(yay)],[A=D,B=C],[A,B,D,C])
22 state([eq(A,B),leq(C,D),test(oh)],[A=D,B=C],[A,B,D,C])
23
24 This critical pair stems from the critical ancestor state:
25 [test(A,B),A=D,B=C,A==B,D>=C]
26
27 with the overlapping part:
28 [ (test(A,B),test(D,C))]
29
30 of the following two rules:
31 test(E,F)<=>E==F|test(yay)
32 test(K,L)<=>K>=L|test(oh)
33 ==========================================================================
34
35 ==========================================================================
36 The following critical pair is not joinable:
37 state([leq(A,B),leq(C,D),test(doh)],[A=D,B=C],[A,B,D,C])
38 state([leq(A,B),leq(C,D),test(oh)],[A=D,B=C],[A,B,D,C])
39
40 This critical pair stems from the critical ancestor state:
41 [test(A,B),A=D,B=C,A=<B,D>=C]
42
43 with the overlapping part:
44 [ (test(A,B),test(D,C))]
45
46 of the following two rules:
47 test(E,F)<=>E=<F|test(doh)
48 test(K,L)<=>K>=L|test(oh)
49 ==========================================================================
```

```
50
51 The CHR program in ~\example7c.pl is NOT confluent!
52 3 non-joinable critical pair(s) found!
53
54 Checking confluence of CHR program in ~\example7c-2.pl...
55
56 ========================================================================
57 The following critical pair is not joinable:
58 state([leq(A,B),eq(C,D),test(doh)],[A=D,B=C],[A,B,D,C])
59 state([leq(A,B),eq(C,D),test(yay)],[A=D,B=C],[A,B,D,C])
60
61 This critical pair stems from the critical ancestor state:
62 [test(A,B),A=D,B=C,B>=A,C==D]
63
64 with the overlapping part:
65 [ (test(A,B),test(D,C))]
66
67 of the following two rules:
68 test(E,F)<=>F>=E|test(doh)
69 test(K,L)<=>L==K|test(yay)
70 ========================================================================
71
72 ========================================================================
73 The following critical pair is not joinable:
74 state([leq(A,B),eq(C,D),test(doh)],[A=D,B=C],[A,B,D,C])
75 state([leq(A,B),eq(C,D),test(oh)],[A=D,B=C],[A,B,D,C])
76
77 This critical pair stems from the critical ancestor state:
78 [test(A,B),A=D,B=C,B>=A,C==D]
79
80 with the overlapping part:
81 [ (test(A,B),test(D,C))]
82
83 of the following two rules:
84 test(E,F)<=>F>=E|test(doh)
85 test(K,L)<=>L==K|test(oh)
86 ========================================================================
87
88 ========================================================================
89 The following critical pair is not joinable:
```

```
state([eq(A,B),eq(C,D),test(yay)],[B=D,A=C],[B,A,D,C])
state([eq(A,B),eq(C,D),test(oh)],[B=D,A=C],[B,A,D,C])

This critical pair stems from the critical ancestor state:
[test(B,A),B=D,A=C,A==B,C==D]

with the overlapping part:
[ (test(B,A),test(D,C))]

of the following two rules:
test(E,F)<=>F==E|test(yay)
test(K,L)<=>L==K|test(oh)
================================================================================

The CHR program in ~\example7c-2.pl is NOT confluent!
3 non-joinable critical pair(s) found!
The input programs are not confluent.
```

7 Conclusion and Future Work

The definition for an operational equivalence check has been implemented and support for more built-ins has been added. This section will first give a short overview and conclusion for all that was done and then talk about what could be done in the future.

7.1 Conclusion

The definition of operational equivalence is very strict and if two programs are operationally equivalent they are usually very similar with only trivial changes. Normally even little changes to a program result in it not being operational equivalent with the original program, but loosening the requirements for equivalence easily leads to a problem that is no longer decidable. Since the operational equivalence checker prints the minimal states that end in not equivalent final states to the scree, the user can check if those special cases actually hurt his program or if they should never actually occur if the program is uses as intended.

Another realization was that the occurring problems did not come from CHR but from Prolog. The main problem was that most built-ins in Prolog cannot work with unbound variables during entailment tests, what is a very important property when trying to check for operational equivalence. The solution for this problem was to move away from the host language and try to represent those built-ins in CHR using CHR constraints and a constraint solver.

The final ressult of this work now is a operational equivalence checker, that works according to Theorem 18. It is limited due to how difficult it is to offer support for further built-ins. The main difficulty when trying to add a built-in is to find a constraint solver that works like the desired built-in, can handle unbound variables and store information for unbound variables.

7.2 Future Work

The limitation to the confluence checker that it cannot handle propagation rules is a big limitation, but to add support for propagation rules the propagation history needs to be considered. To do this one would most likely need to do a lot of code transformations to represent propagation rules with simpagation rules and add an unique identifier to each constraint, as described in section 2.2.2.

The existing constraint solvers that are shown in section 4.2.4 should not be considered perfect representations for those built-ins. Further refinement and additional rules for more special cases could get them closer to the behavior of the built-ins they represent.

Further built-ins could be added by modifying the parser and adding appropriate constraint solvers. But the more built-ins are added the more the other constraint solvers need do be modified since most built-ins have influence on other built-ins (e.g. after adding support for $<$ one needed to ensure that the information that $A \leq B$ and $B < A$ leads to a failed state)

The given operational equivalence checker could be used to implement a operational c-equivalence checker. For this a predicate to find the c-minimal states are needed, then those c-minimal states could be checked for operational equivalence similar to how the minimal states are checked.

A Disc Content

Folder ’./conf+opeq-checker’

This folder contains the operational equivalence checker and the modified confluence checker.

Folder ’./conf+opeq-checker/conf-examples’

This folder contains the example programs shown in section 5.4 and section 6.3 .

Folder ’./conf+opeq-checker/opeq-examples’

This folder contains the example programs shown in section 4.3 .

File ’./conf+opeq-checker/MANUAL’

This file describes how to use the operational equivalence and the confluence checker.

File ’./Bachelor-Thesis-Frank-Richter.pdf’

This file is the pdf version of this work.

Bibliography

[1] *State Equivalence and Confluence Checker*. `http://www.uni-ulm.de/en/in/pm/research/topics/chr/info/downloads.html,` . – Accessed: 2014-05-18

[2] ABDENNADHER, Slim ; FRÜHWIRTH, Thom: Operational equivalence of CHR programs and constraints. In: *Principles and Practice of Constraint Programming–CP'99* Springer, 1999, S. 43–57

[3] FAGES, François ; OLIVEIRA RODRIGUES, Cleyton M. ; MARTINEZ, Thierry: Modular CHR with ask and tell. In: *Proc. of Fifth Workshop on Constraint Handling Rules*, 2008

[4] FRÜHWIRTH, Thom: Constraint Handling Rules by Thom Frühwirth, Cambridge University Press, 2009. Hard cover: ISBN 978-0-521-87776-3. (2009), S. xvii 53–57 59–64 71 101–110 128–132

[5] FRÜHWIRTH, Thom: Introducing Simplification Rules. (1991)

[6] LANGBEIN, Johannes ; RAISER, Frank ; FRÜHWIRTH, Thom: A State Equivalence and Confluence Checker for CHR. In: *7th International Workshop on Constraint Handling Rules*, 2010

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

[8] SNEYERS, Jon ; VAN WEERT, Peter ; SCHRIJVERS, Tom ; DE KONINCK, Leslie: As time goes by: Constraint Handling Rules. In: *TPLP* 10 (2010), Nr. 1, S. 1–47

Name: Frank Richter Matrikelnummer: 626272

Erklärung

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

Ulm, den .....

Frank Richter