Analysis of Constraint Handling Rules Programs

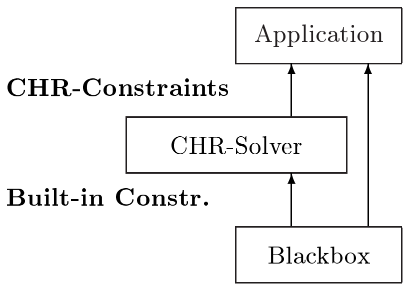
Thom Frühwirth and Slim Abdennadher
Constraint Working Group
Ludwig-Maximilians-University Munich
Department of Computer Science
Oettingenstrasse 67, D-80538 Munich Germany
fruehwir@informatik.uni-muenchen.de
www.pst.informatik.uni-muenchen.de/~fruehwir/cwg.html

Invited Talk at CCL'99

Keine Anmerkung

Constraint Handling Rules (CHR)

Declarative programming language for the specification and implementation of constraint solvers and programs.
CHR constraint solvers are open and flexible, can be maintained, combined, debugged, analysed.



Host language (Prolog, Lisp, Java,...)

[Frühwirth, *Journal of Logic Programming*, 1998]

Anmerkung

Constraint-Programm: schickt nach und nach Constraints an Constraintlöser, erwartet (Teil-)Lösungen.

Constraintlöser: Programm, das Constraints propagiert, vereinfacht und löst.

Built-in Constraints (auch Tests und Zuweisungen)

Vorteile Constraint Handling Rules

- Höhere Programmiersprache
- Deklarative Semantik
- Theoretische Resultate
- Konkrete Anwendungen
- Programmanalyse
- Neue Constraints möglich

- In general constraint application are written in a CLP language using black box solvers. Nevertheless these solvers are efficient, this approach makes it hard to modify a solver or build a solver over a new domain, let alone reason about and analyze it.
- CHR is a proposal to extend CLP languages to allow the user to define new constraints, together with rules specifying how the new constraints react with the constraint store.
- CHR: declarative
- Black-box solver: non-declarative

Example Order Relation $\leq$

```
x ≤x ⇔  true            Reflexivity
x ≤y ∧ y ≤x ⇔ x=y   Antisymmetry
x ≤y ∧ y ≤z ⇒ x ≤z   Transitivity

a ≤b ∧ b ≤c ∧ c ≤a     Query
                        Transitivity
   a ≤c
                        Antisymmetry
          a=c           Blackbox Solver
                        Antisymmetry
   a=b

a=b ∧ a=c               Answer
```


Anmerkung

Constraint aus Zahlenschloß-Beispiel.

`x, y, z` sowie `a, b, c` sind ungebundene Variablen.

Linie zwischen Program und Query ziehen.

Constraint-Problem heißt hier Query,
Lösung bzw. möglichst vereinfachte Form heißt hier Answer.

GLEICH BEISPIEL MACHEN!

Constraints sind (allquantorfreie) Konjunktionen von atomaren Constraints.

CHR-Regeln beschreiben wie Constraints sich zu neuen Constraints vereinfachen und wie Constraints andere Constraints propagieren.

Fortgesetzte Vereinfachung und Propagierung von Constraints führt schließlich zur Lösung der Constraints.

Regeln aus Inferenzregeln, Formeln, Ersetzungsregeln gewinnbar.

Hier Abstrakte Syntax verwendet.

Simplifikation entfernt Constraints, Propagierung nicht.

Kein Beweiser, sondern effizient Konjunktionen lösen.

Beispiel mit Wächter: Reflexivity mit Wächter.

Introductory Example $\leq$

- defined by CHR: $\leq$
- defined by the blackbox solver: *true*, $=$

$$
\begin{array}{rcll}
X \leq X & \Leftrightarrow & true & \text{(reflexivity)} \\
[X \leq Y & \Leftrightarrow & X = Y \mid true & \text{(reflexivity)]} \\
X \leq Y \wedge Y \leq X & \Leftrightarrow & X = Y & \text{(antisymmetry)} \\
X \leq Y \wedge Y \leq Z & \Rightarrow & X \leq Z & \text{(transitivity)}
\end{array}
$$

Remarks

- CHR consists of guarded rules. We distinguish two kinds of rules. Simplification rule replaces constraints by simpler constraints while preserving logical equivalence. Propagation rule adds new constraints, which are logically redundant but may cause further simplification.
- We define a user-defined constraint for less-than-or-equal. The syntactical equality and `true` are built-in constraints. The CHR program implements reflexivity, antisymmetry, and transitivity in a straightforward way.
- The reflexivity rule states that `X≤X` is logically true. Whenever we see a constraint of the form `A≤A` we can simplify it with true. The reflexivity rule can be written using a guard (a precondition on the aplicability of the rule).
- The antisymmetry rule means that if we find `X≤Y` as well as `Y≤X` in the current constraint, we can replace them by the logically equivalent `X=Y`.
- The transitivity rule propagates constraints. It states that the conjunction of `X≤Y` and `Y≤Z` implies `X≤Z`. Operationally, we add the logical consequence `X≤Z` as a redundant constraint.
- Redundancy from propagation rules is useful, as the following computation shows:

Declarative Semantics

Head	H	CHR-Constraints
Guard	C	Built-in Constraints
Body	B	Built-in and CHR-Constraints

Simplification rule

$H \Leftrightarrow C \mid B \qquad \forall\ (C \rightarrow (H \leftrightarrow \exists \bar{x}\ B))$

Propagation rule

$H \Rightarrow C \mid B \qquad \forall\ (C \rightarrow (H \rightarrow \exists \bar{x}\ B))$

Declarative semantics of a CHR program P:

$\mathcal{P}$: Logical meaning of the CHR rules of P.

$\cup\ \mathcal{T}$: Theory for built-in constraints.

Anmerkung

Meta-Variablen stehen immer für Konjunktionen von Constraints.

Variablen $\bar{x}$ kommen *nur* in B vor.

Constraints als spezielle Prädikate der Prädikatenlogik erster Stufe.
Built-in: *true*, *false*, = syntaktische Gleichheit.

CHR-Constraints durch Regeln definiert und implementiert.

Der Wächter als Vorbedingung einer Regel impliziert dabei einen logischen Zusammenhang zwischen Kopf und Rumpf der Regel: Im Fall einer Simplifikationsregel ist der Zusammenhang eine logischen Äquivalenz zwischen Kopf und Rumpf, im Fall einer Propagierungsregel eine Implikation.

Operational Semantics

State Transition System

State: Conjunction of constraints.

Simplify $\quad A \wedge G \ \longmapsto\ B\sigma \wedge G$

if $(H \Leftrightarrow C \mid B) \in P$

and $(A = H\sigma)$ and $(G_V \rightarrow C\sigma)$

Propagate $\quad A \wedge G \ \longmapsto\ B\sigma \wedge A \wedge G$

if $(H \Rightarrow C \mid B) \in P$

and $(A = H\sigma)$ and $(G_V \rightarrow C\sigma)$

Split $\quad (A \vee B) \wedge G \ \longmapsto\ (A \wedge G) \vee (A \wedge G)$

Anmerkung

H, C, B, G: Konjunktionen von Constraints.

Substitution, Matching, Instanz

Jede passende CHR-Regel-Kopie auswählbar.
committed-choice. verpflichtende (Regel-)wahl
Matching anstatt Unifikation (sonst incomplete).
Wächter zur Regelauswahl (da kein backtracking).

$\bar{x}$ frische Variablen aus Regelkopie.

Vereinfachen von built-in Constraints implizit.

Konjunktion assoziativ und kommutativ, nicht idempotent.

$\forall$ Allabschluß.

C im Folgezustand nicht dabei.

Terminierung der Propagierungsregeln nicht dargestellt.

Operational Semantics

State: Conjunction of constraints.

Simplify

If	$(H \Leftrightarrow C \mid B)$ is fresh variant of rule in P
and	G_{pre} are the built-in constraints in G
and	$\mathcal{T} \models G_{pre} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then	$\dfrac{H' \wedge G}{H{=}H' \wedge B \wedge G}$

Propagate

If	$(H \Rightarrow C \mid B)$ is fresh variant of rule in P
and	G_{pre} are the built-in constraints in G
and	$\mathcal{T} \models G_{pre} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then	$\dfrac{H' \wedge G}{H{=}H' \wedge B \wedge H' \wedge G}$

Remarks

- "$\wedge$" is AC (or AC1 with unit $\top$)
- "$H{=}H'$": syntactic equality per component of the conjunctions H and H';
- simplified states and rules;
- Actual states contain information to avoid trivial nontermination of propagation rules.

Correspondence between Semantics

Lemma (Equivalence of States) If the state B appears in a computation of A, then

$$\mathcal{P}, \mathcal{T} \models \forall\ (A \leftrightarrow \exists \bar{x} B).$$

Theorem (Soundness) If C is an answer of A, then

$$\mathcal{P}, \mathcal{T} \models \forall\ (A \leftrightarrow \exists \bar{x} C).$$

Theorem (Completeness) A has at least one finite computation. If $\mathcal{P}, \mathcal{T} \models \forall\ (A \leftrightarrow \exists \bar{x} C)$, then A has an answer C' such that

$$\mathcal{P}, \mathcal{T} \models \forall\ (\exists \bar{x} C \leftrightarrow \exists \bar{y} C').$$

[Abdennadher, Frühwirth, Meuss, *Constraint Journal*, Kluwer, 1999]

Anmerkung

Ableitung: Folge von Reduktionen.

$\exists \bar{x}, y$: Alle Variablen, die nicht in A vorkommen.

Sätze besser als für herkömmliche CLP-Sprachen.

Vollständigkeits- und Korrektheitssätze des CHR-Kalküls basieren vor allem darauf, daß Reduktionen für CHR die logische Äquivalenz von den Zuständen erhalten:

Induktionsbeweis über Länge der Ableitungen plus logische Leseweise der Reduktionsregeln.

Aus diesem Lemma folgt unmittelbar, daß alle Zustände in einer Ableitung einer Query logisch äquivalent sind.

Aus dem Lemma folgt auch die Korrektheit.

Der Vollständigkeitssatz gilt nicht, wenn A keine endlichen Ableitungen hat:

$\mathtt{p} \Leftrightarrow \mathtt{p}$

Sei A die Query $\mathtt{p}$. Es gilt $\mathcal{P}, \mathcal{T} \models \mathtt{p} \leftrightarrow \mathtt{p}$.
Aber A hat nur eine unendliche Ableitung.

Vollständigkeit schwach für Answer *false*:

$\mathtt{p} \Leftrightarrow \mathtt{q}$
$\mathtt{p} \Leftrightarrow false$

Es gilt $\mathcal{P}, \mathcal{T} \models \neg\mathtt{q}$. Aber $\mathtt{q}$ hat nur Answer $\mathtt{q}$.

Keine Anmerkung

CHR at Work

`www.informatik.uni-muenchen.de/~fruehwir/`

7 Implementations

- Eclipse 4.2 Prolog
- Sicstus 3.7 Prolog
- Java (on the way)

30+ Applications

- Theorem Proving, *Uni York, Uni Koblenz*
- Munich rent advisor, *LMU Munich*
- Optimal placement of senders *LMU Munich*
- Executable Z-Specifications, *Uni Melbourne*
- Morphological analysis of natural language, *CIS*
- Integration, Brokering of Internet Information, *MIT*
- Kanji Font Description, *Uni Zurich*
- Automatic Generation of Constraint Solvers, *CWI*
- University Lecture Timetabling, *LMU Munich*

Keine Anmerkung

CHR Constraint Solvers

50+ Constraint Solvers

- Boolean Algebra, Propositional Logic, *Uni York*
- Herbrand Terms, Feature Terms, Types, Lists, Sets
- Finite Domains
- Linear Polynomial (In-)Equations
- Nonlinear Polynomial (In-)Equations, *CRIN-CNRS, INRIA-Lorraine*
- Interval Arithmetic
- Generic Arc and Path Consistency
- Temporal Reasoning, CISE Milano, *Uni Linkoeping*
- Spatial Reasoning, *Uni Jaume I Casellun*
- Terminological Reasoning, *DFKI Kaiserslautern, Uni Bucharest, GMD FIRST Berlin*
- Diagrammatical Reasoning, *Uni Melbourne*

Comparison

CHR	Prolog	TRS
Relational	Relational	Functional
Simplification rules	Clauses	Rewrite rules
Propagation rules	-	-
Multi-headed rules	-	AC
Committed-choice	Backtracking	Committed-choice
Don't care nondeterm.	Don't know	Don't care
Matching	Unification	Matching
Guards	-	Conditions

Anmerkung

committed-choice bedingt:

Matching anstatt Unifikation (sonst incomplete).

Wächter zur Regelauswahl (da kein backtracking).

Furthermore:
CHR-TRS
Multiple occurences of variables in head
local variables
global knowledge through built-in constraint store

CHR Program Analysis

Consistency

The logical meaning of the rules is consistent.

Confluence

The answer of a query is always the same, no matter which of the applicable rules are applied.

Completion

Makes non-confluent programs confluent by adding new rules.

Equivalence

Are CHR programs and constraints operationally equivalent?

Keine Anmerkung

Program Analysis

Confluence

The answer of a query is always the same, no matter which of the applicable rules are applied.

```
x ≤y ∧ y ≤x ⇔ x=y
x ≤y ∧ y ≤z ⇒ x ≤z

x ≤y ∧ y ≤x

x ≤y ∧ y ≤x ∧ x ≤x      x=y

x=y ∧ x ≤x         x=y
```

Theorem (Confluence) A terminating CHR program is confluent iff its critical pairs are joinable.

[Abdennadher, Frühwirth, Meuss, *Constraint Journal*, Kluwer, 1999]

Anmerkung

A, B, C, D sind Zustände, d.h. Constraints.

Probleme bei der Adaptation von Ergebnissen aus Termersetzungssystemen (TRS):

- Propagationsregeln
- Blackbox-Constraintlöser

Monotonie entspricht *Verträglichkeit* bei TRS: Wenn $A \longmapsto B$, dann auch $A \wedge D \longmapsto B \wedge D$.

Überlappen zweier Regeln ergibt Ausgangszustand.

Ausgangszustand hat die Köpfe der beiden Regeln, wobei ein oder mehrere Kopfconstraints miteinander gleichgesetzt wurden, und die Wächter der beiden Regeln.

Entscheidbares, hinreichendes und notwendiges Kriterium.

Nutzen des Confluencetests:

Theorie:
Impliziert Konsistenz und verbessert Übereinstimmung der Semantics.

Praxis:
Answer auch unabhängig von Reihenfolge der Constraints.

Confluencetest enthüllt Fehler.
Implementierte CHR-Constraintlöser sind meist confluent.

Vervollständigung kann nicht-confluente Programme confluent machen.

Confluence

Given a goal, every computation leads to the same result no matter what rules are applied.

A decidable, sufficient and necessary condition for confluence of terminating CHR programs through joinability of critical pairs.

Example $\leq$

$$
\begin{array}{rcll}
X \leq X & \Leftrightarrow & \mathit{true} & \text{(reflexivity)} \\
X \leq Y \wedge Y \leq X & \Leftrightarrow & X = Y & \text{(antisymmetry)}
\end{array}
$$

$$
\begin{tikzcd}
 & A \leq A \wedge A \leq A \arrow[dl, "\text{reflexivity}"'] \arrow[dr, "\text{antisymmetry}"] & \\
A \leq A \arrow[dr, "\text{reflexivity}"'] & & A = A \arrow[dl] \\
 & \mathit{true} &
\end{tikzcd}
$$

Remarks

- In previous work we introduced a notion of confluence for CHR programs. Confluence is an essential syntactical property of any constraint solver. It ensures that the solver will always compute the same result for a given set of constraints independent of which rules are applied.
- We gave a decidable, sufficient and necessary syntactic condition for confluence of terminating CHR programs. This condition adopts the notion of critical pairs as known from term rewriting systems.
- These critical pairs can be derived from rules with overlapping heads.
- Consider the program for less-than-or-equal. The critical state stems from unifying the first atom of the head of the reflexivity rule with the first atom of the head of the antisymmetry rule. The critical pair stems from applying the reflexivity and the antisymmetry rule.

Confluence

Given a goal, every computation leads to the same result no matter what rules are applied.

A decidable, sufficient and necessary condition for confluence of terminating CHR programs through joinability of critical pairs.

Example `max`

```
max(X,Y,Z) ⇔  X≤Y |  Z=Y.      (r1)
max(X,Y,Z) ⇔  X≥Y |  Z=X.      (r2)
```

$$\begin{tikzcd}
& \mathtt{max(X,Y,Z)} \wedge \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \geq \mathtt{Y} \arrow[dl, "\mathtt{r1}"'] \arrow[dr, "\mathtt{r2}"] & \\
\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \geq \mathtt{Y} \wedge \mathtt{Z} = \mathtt{Y} \arrow[dr] & & \mathtt{X} \leq \mathtt{Y} \wedge \mathtt{X} \geq \mathtt{Y} \wedge \mathtt{Z} = \mathtt{X} \arrow[dl] \\
& \mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z} &
\end{tikzcd}$$

Remarks

- In previous work we introduced a notion of confluence for CHR programs. Confluence is an essential syntactical property of any constraint solver. It ensures that the solver will always compute the same result for a given set of constraints independent of which rules are applied.
- We gave a decidable, sufficient and necessary syntactic condition for confluence of terminating CHR programs. This condition adopts the notion of critical pairs as known from term rewriting systems.
- These critical pairs can be derived from rules with overlapping heads.
- Consider the program for `max`. The critical state stems from unifying the the head of first rule `r1` with the head of the second rule `r2`. The critical pair stems from applying `r1` and `r2`.

Why Confluence?

Given a goal, every computation leads to the same result no matter what rules are applied.

$$\begin{tikzcd}
& S \arrow[dl, "*"'] \arrow[dr, "*"] & \\
S_1 \arrow[dr, "*"'] & & S_2 \arrow[dl, "*"] \\
& T &
\end{tikzcd}$$

- Theoretical: implies consistency and improves soundness and completeness.
- Practical: Non-confluent CHR programs represent ill-defined constraint solvers.

Example: Program with typo:

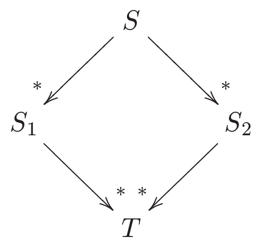
```
max(X,Y,Z) ⇔ X=<Y | Y=Z.
max(X,Y,Z) ⇔ Y=<X | X=Y.
```

`max(1,1,0)` is either `false` or `true`.

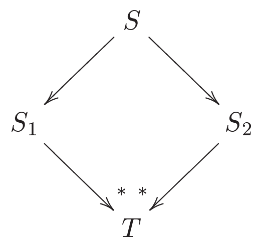
No Remarks

Local Confluence

Confluence



Local Confluence



Termination + Local Confluence = Confluence.

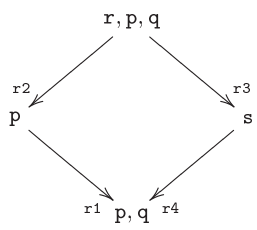
Confluence check: Infinitely many pairs to test!

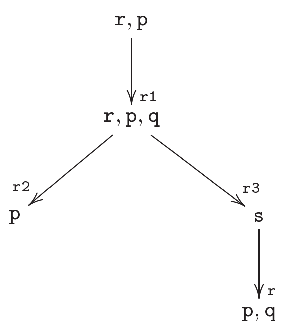
Idea (TRS): Test only minimal, "critical" pairs.

No remarks

Confluence of Propagation Rules

```
r1 @ p ⇒ q.
r2 @ r,q ⇔ true.
r3 @ r,p,q ⇔ s.
r4 @ s ⇔ p,q.
```





Violated Confluence

Remarks

The *token store* contains rulenames together with the corresponding conjunctions of user-defined constraints that unify with the heads of the respective propagation rules. The updating of the token store ($T_{(C,C_U)}$) by introducing user-defined constraints depends on the introduced constraints C and the current user-defined store C_U.

Critical Pairs

$$\text{keep}(R) := \begin{cases} \text{true}, & \text{if } R \text{ simplification rule} \\ H\ , & \text{if } R \text{ propagation rule w. head } H \end{cases}$$

Critical pairs derive from overlapping rules:

$$R \text{ @ } H_1 \wedge H_2 \rightleftharpoons G \mid B$$
$$R' \text{ @ } H_1' \wedge H_2' \rightleftharpoons G' \mid B'$$
where H_1 and H_1' unify.

The critical ancestor state is:

$$H_1 \wedge H_2 \wedge H_2' \wedge H_1{=}H_1' \wedge G \wedge G'$$

The critical pair is:

$$\left(\begin{array}{c} \text{keep}(R) \wedge H_2' \wedge H_1{=}H_1' \wedge G \wedge G' \wedge B, \\ \text{keep}(R') \wedge H_2 \wedge H_1{=}H_1' \wedge G \wedge G' \wedge B' \end{array} \right)$$

A critical pair (S_1, S_2) is joinable, iff the states S_1 and S_2 are joinable.

No remarks

Completion

Derive rules from a non-joinable critical pair that would allow a transition from one of the critical states into the other one.

$$\begin{array}{lcll} X \leq Y \wedge Y \leq X & \Leftrightarrow & X = Y & \text{(antisymmetry)} \\ X \leq Y \wedge Y < X & \Leftrightarrow & false & \text{(inconsistency)} \end{array}$$

$$\begin{tikzcd} & A \leq B \wedge B \leq A \wedge B < A \arrow[dl, "\text{antisymmetry}"'] \arrow[dr, "\text{inconsistency}"] & \\ A = B \wedge B < A \arrow[d] & & B \leq A \wedge false \arrow[d] \\ A = B \wedge A < A & & false \end{tikzcd}$$

$$X < X \quad \Leftrightarrow \quad X = Y \quad \mid \quad false \quad \text{(irreflexivity)}$$

Remarks

- The idea of completion is to derive a rule from a non-joinable critical pair that would allow a transition from one of the critical states into the other one, thus re-introducing confluence.
- This example shows that the completion method can be used - to some extent – to specialize constraints.
- We extend the CHR program for less-than-or-equal by a simplification rule expressing the interaction between less-than-or-equal and less. Then the resulting program loses confluence.
- The completion procedure inserts the following rule ... expressing the irreflexivity of $<$.

Generation of Rules

$$
\begin{array}{llll}
p(X,Y) & \Leftrightarrow & X \geq Y \wedge q(X,Y) & \text{(r1)} \\
p(X,Y) & \Leftrightarrow & X > Y \wedge r(X,Y) & \text{(r2)}
\end{array}
$$

$$
\begin{tikzcd}
 & p(A,B) \arrow[dl, "\text{r1}"'] \arrow[dr, "\text{r2}"] & \\
q(A,B) \wedge A \geq B & & r(A,B) \wedge A > B
\end{tikzcd}
$$

$$
\begin{array}{llllll}
r(X,Y) & \Leftrightarrow & X > Y & \mid & q(X,Y) \wedge X \geq Y & \text{(r3)} \\
q(X,Y) & \Rightarrow & X \geq Y & \mid & X > Y & \text{(r4)}
\end{array}
$$

Remarks

- In contrast to completion methods for TRS, we need more than one rule to make a critical pair joinable and in general it is not sufficient to insert only simplification rules as in completion for TRS, in order to join a non-joinable critical pair.

Inference Rules of Completion

C = set of critical pairs, P = set of rules.

CP-Deduction:

$$
\frac{(C,P) \qquad (S_1,S_2) \text{ is a critical pair of } P}{(C \cup \{(S_1,S_2)\}, P)}
$$

CP-Simplification:

$$
\frac{(C \cup \{(S_1,S_2)\}, P) \qquad S_1 \mapsto S_1' \qquad S_2 \mapsto S_2'}{(C \cup \{(S_1',S_2)\}, P)}
$$

CP-Deletion:

$$
\frac{(C \cup \{(S_1,S_2)\}, P) \qquad S_1 \text{ and } S_2 \text{ are joinable}}{(C,P)}
$$

CP-Orientation:

$$
\frac{(C \cup \{(S_1,S_2)\}, P) \qquad R = \mathit{orient}_{\gg}(S_1,S_2)}{(C, P \cup R)}
$$

Remarks

- Our completion algorithm maintains a set C of critical pairs and a set P of rules.
- The rule *CP-Deduction* permits to add critical pairs to C.
- The rule *CP-Simplification* replaces state in a critical pair by its successor state.
- The rule *CP-Deletion* removes a joinable critical pair.
- *CP-Orientation* removes a critical pair from C and adds new rules to P, provided the critical pair can be oriented with respect to the termination ordering $\gg$.

1. The algorithm stops successfully
2. The algorithm aborts unsuccessfully (Cons. Theorem)
3. The algorithm does not terminate

Correctness Theorem

If P is a range-restricted terminating CHR program and the completion procedure is successful

Then: Output P'

- confluent
- terminating
- logically equivalent to P

Consistency Theorem

If the completion procedure aborts unsuccessfully because the final states have only built-in constraints that differ, then the logical meaning of P is inconsistent.

$$
\begin{array}{rcll}
X \leq X & \Leftrightarrow & \mathit{false} & \text{(reflexivity)} \\
X \leq Y \wedge Y \leq X & \Leftrightarrow & X = Y & \text{(antisymmetry)}
\end{array}
$$

$$
\begin{tikzcd}
 & A \leq A \wedge A \leq A \arrow[dl, "\text{reflexivity}"'] \arrow[dr, "\text{antisymmetry}"] & \\
false \wedge A \leq A \arrow[d] & & A = A \arrow[d] \\
false & & true
\end{tikzcd}
$$

Remarks

- Another property of the completion procedure is that it can exhibit inconsistency of the program to complete.
- The logical meaning of this program is not a consistent theory.

Conclusion

- Completion method for Constraint Handling Rules to make a non-confluent CHR program confluent by adding new rules.
- Completion helps the CHR programmer to extend, modify and specialize existing solvers instead of having to write them from scratch.

Current Work: The relationship of completion to partial evaluation.

Operational Equivalence

- Correctness of program transformation
- The use of modules or libraries with similar functionality
- Combination of constraint solvers

Example: Are the two CHR rules defining `max`

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

operationally equivalent to these two rules?

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


No Remarks

No Remarks

Compatibility of Programs

Let P_1 and P_2 be two confluent and terminating CHR programs and let the union of the two programs, $P_1 \cup P_2$, be terminating. P_1 and P_2 are *compatible* if $P_1 \cup P_2$ is confluent.

Example `max`

$P1$: `max(X,Y,Z)` $\Leftrightarrow$ `X<Y` | `Z=Y.`
`max(X,Y,Z)` $\Leftrightarrow$ `X`$\geq$`Y` | `Z=X.`

$P2$: `max(X,Y,Z)` $\Leftrightarrow$ `X`$\leq$`Y` | `Z=Y.`
`max(X,Y,Z)` $\Leftrightarrow$ `X>Y` | `Z=X.`

3 critical ancestor states coming from one rule in P_1 and one rule in P_2:

- `max(X,Y,Z)` $\wedge$ `X<Y` $\wedge$ `X`$\leq$`Y`
- `max(X,Y,Z)` $\wedge$ `X`$\geq$`Y` $\wedge$ `X`$\leq$`Y`
- `max(X,Y,Z)` $\wedge$ `X`$\geq$`Y` $\wedge$ `X>Y`

No Remarks

Equivalence of CHR Programs

Let P_1 and P_2 be CHR programs.

A state S is P_1, P_2*-joinable*, iff there are two computations $S \mapsto^*_{P_1} T$ and $S \mapsto^*_{P_2} T$, where T is a final state.

P_1 and P_2 are *operationally equivalent* if all states are P_1, P_2-joinable.

The *set of critical states of* P_1 *and* P_2 is defined as follows:

$$\{H \wedge C \mid (H \odot C \mid B) \in P_1 \cup P_2, \odot \in \{\Leftrightarrow, \Rightarrow\}\}$$

Theorem Let P_1 and P_2 be terminating and confluent programs. P_1 and P_2 are operationally equivalent iff all critical states of P_1 and P_2 are P_1, P_2-joinable.

Example `range`

`range(X,Min,Max)`: `X` is between `Min` and `Max`.

P_1:

`range(X,Min,Max)` $\Leftrightarrow$ `max(X,Min,X)` $\wedge$ `max(X,Max,Max).`

P_2:

`range(X,Min,Max)` $\Leftrightarrow$ `Max<Min` | `false.`
`range(X,Min,Max)` $\Leftrightarrow$ `Min≤Max` | `Min≤X` $\wedge$ `X≤Max.`

P_1 and P_2 are not operationally `range`-equivalent.
`range(X,Min,Max)` is not P_1, P_2-joinable:

$\mathtt{range}(\mathtt{X}, \mathtt{Min}, \mathtt{Max}) \mapsto_{P_1}$
$\mathtt{max}(\mathtt{X}, \mathtt{Min}, \mathtt{X}) \wedge \mathtt{max}(\mathtt{X}, \mathtt{Max}, \mathtt{Max})$

In P_2, $\mathtt{range}(\mathtt{X}, \mathtt{Min}, \mathtt{Max})$ is a final state.

P_1 is "operationally stronger" than P_2:

In P_1, `range(5,6,Max)` is inconsistent.

In P_2, `range(5,6,Max)` is a final state.

Equivalence of CHR Constraints

Let c be a CHR symbol. A *c-state* is a state where all user-defined constraints have the same CHR symbol c.

Let c be a CHR symbol defined in two CHR programs P_1 and P_2. P_1 and P_2 are *operationally c-equivalent* if all c-states are P_1, P_2-joinable.

The set of *c-critical states* is defined as follows:

$\{H \wedge C \mid (H \odot C \mid B) \in P_1 \cup P_2, \odot \in \{\Leftrightarrow, \Rightarrow\}$
and H contains only c-dependent CHR symbols$\}$

Theorem Let c be a CHR symbol defined in two confluent and terminating CHR programs P_1 and P_2. P_1 and P_2 are operationally c-equivalent if all c-critical states are P_1, P_2-joinable.

Example sum

`sum(List,Sum)`: `Sum` is the sum of elements of `List`.

P_1:

```
sum([],Sum) ⇔ Sum=0.
sum([X|Xs],Sum) ⇔ sum(Xs,Sum1) ∧ Sum=Sum1+X.
```

P_2:

```
sum([],Sum) ⇔ Sum=0.
sum([X|Xs],Sum) ⇔ sum1(X,Xs,Sum).
sum1(X,[],Sum) ⇔ Sum = X.
sum1(X,Xs,Sum) ⇔ sum(Xs,Sum1) ∧ Sum=Sum1+X.
```

The `sum`-critical states
`sum([],Sum)` and `sum([X|Xs],Sum)`
are P_1, P_2-joinable:

Final state of `sum([],Sum)` is `Sum=0`.

$$\begin{array}{lll} \texttt{sum([X|Xs],Sum)} & \mapsto_{P_1} & \texttt{sum}(\texttt{Xs},\texttt{Sum1}) \wedge \texttt{Sum} = \texttt{Sum1} + \texttt{X} \\ \texttt{sum([X|Xs],Sum)} & \mapsto_{P_2} & \texttt{sum1}(\texttt{X},\texttt{Xs},\texttt{Sum}) \\ & \mapsto_{P_2} & \texttt{sum}(\texttt{Xs},\texttt{Sum1}) \wedge \texttt{Sum} = \texttt{Sum1} + \texttt{X} \end{array}$$

Remarks

- Operational equivalence $\Longrightarrow$ Compatibility
- Compatibility $\not\Longleftarrow\!\!\!\Longrightarrow$ Operational equivalence
- Operational equivalence of two CHR programs $\Longrightarrow$ operational c-equivalence of all common constraints c
- Operational c-equivalence of all common constraints of two CHR programs $\not\Longrightarrow$ Operational equivalence

P_1:

```
p ⇔ s.
s ∧ q ⇔ true.
```

P_2:

```
p ⇔ s.
s ∧ q ⇔ false.
```

Three common CHR symbols: `p`, `s` and `q`.

- `s` and `p` are the `p`-dependent constraint symbols.
- There are no `s`-dependent CHR symbols except `s` itself.
- Analogously for `q`.

`p` is the only `p`-critical state. It is P_1, P_2-joinable.
P_1 and P_2 are not operationally equivalent:

- $\texttt{s} \wedge \texttt{q} \mapsto_{P_1} \texttt{true}$
- $\texttt{s} \wedge \texttt{q} \mapsto_{P_2} \texttt{false}$

Boolean Algebra, Propositonal Logic

Truth values are 1 and 0, logical connectives.

```
X and Y = Z <=> X=0 | Z=0.
X and Y = Z <=> X=1 | Y=Z.
X and Y = Z <=> Z=1 | X=1,Y=1.
X and Y = Z <=> X=Y | Y=Z.
```

E. Dumbill (A. Frisch), Univ. of York, 1995
Theorem Proving, variants of the
Davis-Putnam procedure.

Anmerkung

Boolesche Constraints leiten sich aus der Booleschen Algebra (auch: Schaltalgebra) ab. Die Constraints des Constraintsystems B sind logische Formeln über Variablen, die nur die Werte 1 oder 0 annehmen können, z.B. ($X \sqcup Y =_B 1$). Diese Booleschen Variablen werden in der Literatur auch als binäre (Schalt-)Variablen bezeichnet.

Die Signatur von B enthält die (Booleschen) Konstanten 1 und 0 (sie stehen logisch für die Wahrheitswerte *wahr* und *falsch* bzw. für die Spannungstufen *H(igh)* und *L(ow)* digitaler Schaltungen), logische Operatoren als Funktionssymbole, z.B. die einstellige Negation $-$ und die zweistelligen Verknüpfungen $\sqcap$ (und), $\sqcup$ (oder) und $\oplus$ (exklusives Oder), sowie die Constraintsymbole $=_B$, *true* und *false*.

Theorem Proving

Resolution

```
cl(L1), cl(L2) ==> in(+X,L1),in(-X,L2) |
                   append(L1,L2,L), cl(L).
```

Unit Resolution, Davis-Putnam Procedure

```
empty_clause @ cl([]) <=> false.
tautology @ cl(L) <=> in(+X,L),in(-X,L) | true.

unit_instantiation @ cl([+X]) <=> X=1.
unit_instantiation @ cl([-X]) <=> X=0.

unit_propagation @ cl(L) <=> in(+0,L) | cl(L).
unit_propagation @ cl(L) <=> in(-1,L) | cl(L).
unit_subsumption @ cl(L) <=> in(+1,L) | true.
unit_subsumption @ cl(L) <=> in(-0,L) | true.
```


Anmerkung

In [Dum95] experiments were performed in applying resolution and backtracking to solving Boolean constraint satisfaction problems. The DP procedure restricts resolution to unit clauses. A labeling phase is added that tries truth values using backtracking for the variables one by one, thus retaining completeness.

Here is an incremental version of the DP procedure. "Pure literal deletion" is not implemented, because it is based on a global condition which is not sound anymore when constraints can be added incrementally as is the case in CHR.

Boolean CSPs are modeled as conjunctions of clauses, where a clause is a disjunction of literals (positive or negative atomic propositions). A clause is represented as a list of signed Boolean variables. For example, $\neg a \vee b \vee c$ is represented as $[-A, +B, +C]$. Variables in the lists are ordered. in/2 detects membership a list.

Terminological Reasoning, Description Logics

Terminology: Unary concepts and binary roles with logical connectives.

```
parent isa human and every child is human.
father isa parent and man.
```

```
I:C  <=> (C isa S), I:S.
I:S and T  <=> I:S, I:T.
I:nota S, I:S  <=> false.
I:every R is S, (I,J):R  ==> J:S.
```

A. Wolf, GMD FIRST, Berlin, 1996:
ConTeS: Configuration of technical processes with TRLC: executable specification language.

Keine Anmerkung

Keine Anmerkung

Path Consistency, Temporal and Spatial Reasoning

Local concistency method from artificial intelligence.

```
c(I,K,C1),c(K,J,C2) ==>
      composition(C1,C2,C3),  c(I,J,C3).
c(I,J,C1),c(I,J,C2) <=>
      intersection(C1,C2,C3), c(I,J,C3).
```

F. Toledo, Univ. Castellon, Spanien, 1996:
Qualitative spatial reasoning, book on robot motion planning 1999.

M. Bjareland (E. Sandewall), Univ. Linköpping, 1996:
PMON, logic that models dynamic systems with observations, actions, temporal constr.

Keine Anmerkung

Finite Domains, Interval Arithmetic

```
X in N1:M1, X in N2:M2 <=>
   X in max(N1,N2):min(M1,M2).
X≤Y, X in NX:MX, Y in NY:MY ==>
   X in NX:MY, Y in NX:MY.
X+Y=Z, X in NX:MX, Y in NY:MY ==>
   Z in NX+NY:MX+MY, X in NZ-MY:MZ-NY,...
```

M. Duerst, ETH Zürich, 1996:
Analysis and generation of Kanji characters.

M. Marte, S. Abdennadher, LMU, 1997:
Timetabling for university lectures.

Linear Polynomial Equations

Variable Elimination.

```
empty @ B = 0 <=> zero(B).

eliminate @
A1*X+P1 = 0, A2*X+P2 = 0 <=>
   compute(P2+P1*A2/A1,P3),
   A1*X+P1 = 0, P3 = 0.
```

E. Monfroy (M. Rusinowitch), Nancy, 1996:
GroAk - combined with Groebner Bases.

Anmerkung

The `empty` rule says that if the polynomial contains no more variables, the constant `B` must be (approximate to) zero. The `eliminate` rule performs variable elimination. It takes two equations that start with the same variable. The first equation is left unchanged, it is used to eliminate the occurrence of the common variable in the second equation. The auxiliary built-in constraint `compute` simplifies a polynomial arithmetic expression into a new polynomial. Note that no variable is made explicit, i.e. no pivoting is performed. Any two equations with the same first variable can react with each other. Therefore, the solver is highly concurrent and distributed.

The solver can be extended by a few rules to create explicit variable bindings, to make implicit equalities between variables explicit, to deal with inequations using slack variables or fouriers algorithm.

Conclusions

Constraint Handling Rules
Improve on theoretical and practical aspects of constraint programming and solving.

- Executable specification and rapid prototyping. Short, efficient programs.

- More flexible than black box solvers and special tools. Constraint solvers can be modified and combined.

- Enables new application domains with new kinds of constraints.

- Good correspondence between declarative and operational semantics.

- Decidable confluence test exhibits errors and guarantees consistency.

- Completion makes programs confluent and exhibits inconsistencies. Allows for automatic specialisation and modification.

- Operational equivalence can be testet to help combination.

Keine Anmerkung

Conclusions

Constraint Programming

Generic Framework for
- modelling with incomplete information
- solving of combinatorial problems

in decision support systems.

Constraint Handling Rules

Declarative language for constraint programming.
- Executable specifications.
- Good theoretical properties.
- Implementations and libraries available.

Flexible and maintainable intelligent software.

No Remark

Related Work

- Logic for Problem Solving, Kowalski '79
- ALPS language, Maher '87
- CHIP Demons, Forward rules, v. Hentenryck '88
- Bertrand rewrite rules, Leler '88
- Chemical Abstract Machine, Banatre '88
- cc framework, Saraswat '89
- Andorra languages, Haridi '90
- LAURE rules, Caseau '91
- cc(FD) Constr. combinators, v. Hentenryck '91
- **Constraint Handling Rules, Frühwirth '91**
- Guarded rules, Smolka '91
- BNR-Prolog Meta-, reified constraints, Older '92
- Attributed variables, Holzbaur '92
- clp(FD) Indexicals, Codognet '93
- OZ language, Smolka '93
- ELAN rewriting logic, Kirchner '93

Constraint Programming

Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

[Eugene C. Freuder, *Constraints Journal*, 1997]

Anmerkung

60ties,70ties Constraint networks in artificial intelligence.
70ties Logic programming (Prolog).
80ties Constraint logic programming.
80ties Concurrent logic programming.
90ties Concurrent constraint programming.
90ties Commercial applications.

Constraint Handling Rules

For the theoretician meta-theorems can be proved and analysis techniques invented once and for all; for the implementor different constructs can be implemented once and for all; for the user only one set of ideas need to be understood, though with rich (but disciplined) variations (constraint systems).

[P. van Hentenryck, V.A. Saraswat, Strategic Directions in Constraint Programming, *ACM Computing Surveys*, 1996]

Keine Anmerkung