Using Program Analysis for Integration and Optimization of Rule-based Constraint Solvers

Slim Abdennadher and Thom Frühwirth

Computer Science Department, University of Munich
Oettingenstr. 67, 80538 München, Germany
{abdennad,fruehwir}@informatik.uni-muenchen.de

RÉSUMÉ.

ABSTRACT. *One lesson learned from practical applications is that constraints are often heterogeneous. Solving such constraints requires a collaboration of constraint solvers. In this paper, we introduce a methodology for the tight integration of CHR constraint solver programs into one such program. CHR is a high-level rule-based language for writing constraint solvers and reasoning systems. A constraint solver is well-behaved if it is terminating and confluent. When merging constraint solvers, this property may be lost. Based on previous results on CHR program analysis and transformation we show how to utilize completion to regain well-behavedness. We identify a class of solvers whose union is always confluent and we show that for preserving termination such a class is hard to find. The merged and completed constraint solvers may contain redundant rules. Utilizing the notion of operational equivalence, which is decidable for well-behaved CHR programs, we present a method to detect redundant rules in a CHR program.*

MOTS-CLÉS :

KEYWORDS: *Rule-based Constraint Solvers, Cooperation, Combination, Optimization, Program Analysis*

1. Introduction

Many applications of constraint-based reasoning involve heterogeneous constraints. Solving such constraints requires a collaboration of two or more constraint solvers. In this paper, we are concerned with solvers written in CHR language.

CHR (Constraint Handling Rules) [FRÜ 98] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that manipulate conjunctions of constraints. In CHR, we distinguish two kinds of rules : simplification rules replace constraints by simpler constraints. Propagation rules add new constraints which may cause further simplification.

Usually, CHR solvers are *well-behaved*, i.e. terminating and confluent. Confluence means that it does not matter for the result which of the applicable rules are applied in a computation. Once termination has been established [FRÜ 00], there is a decidable, sufficient and necessary test for confluence [ABD 97]. Confluence also implies *consistency* of the logical reading of the solver program [ABD 97].

Given two well-behaved constraint solver CHR programs, then their tight integration is simply the union of their rules. There is no restriction on the signature of the solvers. In particular, solvers may fully or partially define the same constraints. Any computation that was possible in one of the solvers will also be possible in the union of the solvers, since additional rules cannot inhibit the application of old rules (as can be seen from the operational semantics of CHR).

However, the union of the solvers could loose termination and/or confluence, and thus their well-behavedness.

Example 1.1 Consider a solver program P_1 with the single simplification rule $\{\texttt{a} \Leftrightarrow \texttt{b}\}$ that replaces the CHR constraint `a` by the constraint `b` and a solver program P_2 with the single rule $\{\texttt{b} \Leftrightarrow \texttt{a}\}$ that replaces the CHR constraint `b` by the constraint `a`. The union of P_1 and P_2, $\{\texttt{a} \Leftrightarrow \texttt{b}, \texttt{b} \Leftrightarrow \texttt{a}\}$, is obviously non-terminating.

Consider a program P_1 with the single rule $\{\texttt{a} \Leftrightarrow \texttt{b}\}$ and a program P_2 with the single rule $\{\texttt{a} \Leftrightarrow \texttt{c}\}$. Their union $\{\texttt{a} \Leftrightarrow \texttt{b}, \texttt{a} \Leftrightarrow \texttt{c}\}$ is terminating, but obviously non-confluent, since a computation for `a` may result in either `b` or `c` depending on the (committed) choice of the rule.

While establishing termination for CHR programs without propagation rules is in practice often rather simple [FRÜ 00], termination is in general undecidable for CHR programs. On the other hand, *completion* can make non-confluent programs confluent [ABD 98]. Thus there is a chance to automatically produce from two well-behaved constraint solvers a solver that behaves well, too.

Example 1.2 Consider the union of P_1 and P_2 of Example 1.1, $\{\texttt{a} \Leftrightarrow \texttt{b}, \texttt{a} \Leftrightarrow \texttt{c}\}$. To make the union confluent, the rule $\texttt{b} \Leftrightarrow \texttt{c}$ can be added.

In the paper, we also consider the special case of so-called *non-overlapping* solvers that define different constraints. Non-overlapping solvers may have common (shared) CHR constraints and function symbols and have common built-in constraints. We prove that they are well-behaved if their union is terminating. While confluence is modular (preserved) for well-behaved, non-overlapping solvers, we will argue that it is very hard to find a syntactic class of solver programs that admits modularity for termination.

In practice, non-overlapping solvers are integrated using so-called *bridge rules* between the different constraints they define. These bridge rules often destroy well-behavedness and we show by example how completion fares with such solvers.

The resulting constraint solver may contain *redundant rules*. Based on the operational equivalence notion [ABD 99a], we present a method to detect and remove redundant rules in a CHR constraint solver.

Related Work. There is a renewed interest in languages and models for constraint solver cooperation. An overview of the issues in cooperative constraint solving can be found in [GRA 01]. Recent work in this area includes BALI [MON 00], a scheme for integrating heterogeneous solvers by encapsulation : a cooperation language based on strategies is compiled into solver specific communication code. Similarly, the framework of [CAS 00] relies on strategies to specify when component solvers are to be applied. The framework of [HOF 00] requires specific interfaces from the constraint solvers and a meta constraint solver to coordinate the cooperating solvers. Examples and implementations of this framework concentrate on numerical constraints.

When CHR is used as an implementation language for constraint solvers, desirable properties like confluence and operational equivalence can be decided once termination has been established. There is no need for specific interfaces, because the constraint solvers communicate freely via shared variables using their common built-in constraints. In well-behaved CHR solvers, it does not matter which of the applicable rules are applied. In particular, in well-behaved merged solvers, it does not matter from which solvers the rules are coming from. Thus any type of cooperation strategies [GRA 01], be it hard-coded, be it based on priorities or explicit operators, is possible. Moreover, the strategies can be very fine-grained, at the level of the application of a single rule from the solver program, i.e. single computation step.

The work of [TIN 98, BAA 01] focuses on building a constraint solver for the union of theories with given solvers. These theories are usually cast as equational theories. In [BAA 01], the theories are assumed to be disjoint. In [TIN 98], combination of theories sharing constructors have been investigated. In CHR, equalities referring to distinct theories are assumed to be represented by different constraint symbols. CHR programs represent first-order theories, that can be unioned without any requirements. Operationally, however, we want to make sure that the resulting solver is still well-behaved.

Outline of Paper. In Section 2, we define the CHR language and summarize previous results on confluence, completion, and operational equivalence. In the next section of the paper, we show how to merge CHR constraint solvers utilizing completion. We then investigate when termination and confluence are preserved under union of solver programs. We consider the special case of so-called non-overlapping solvers that define different constraints and introduce the notion of so-called bridge rules to integrate such solvers. In Section 6, we show how to remove redundant rules from a solver utilizing operational equivalence.

2. Preliminaries

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) as well as previous results on confluence, completion, and operational equivalence. More detailed presentations can be found in [ABD 97, ABD 99b, ABD 98, ABD 99a].

2.1. *Syntax of CHR*

We use two disjoint sets of predicate symbols for two different kinds of constraints : *built-in constraint symbols* and *CHR constraint symbols (user-defined symbols)*. We call an atomic formula with a constraint symbol a *constraint*. Built-in constraints are handled by predefined given constraint black-box solvers. We assume that these solvers are well-behaved. Built-in constraints include $=$, $true$, and $false$. The semantics of the built-in constraints is defined by a consistent first-order *constraint theory* CT. In particular, CT defines $=$ as the syntactic equality over finite terms.

CHR constraints are defined by a CHR program.

Definition 2.1 A *CHR program* is a finite set of rules. There are two kinds of rules : A *simplification rule* is of the form $Name \ @ \ H \Leftrightarrow C \mid B$. A *propagation rule* is of the form $Name \ @ \ H \Rightarrow C \mid B$, where *Name* is an optional, unique identifier of a rule, the *head* H is a non-empty conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a conjunction of built-in and CHR constraints. A guard "*true*" is often omitted together with the commit symbol $\mid$.

A CHR symbol is *defined* in a CHR program if it occurs in the head of a rule in the program.

Example 2.1 We define a CHR constraint for a partial order relation $\leq$:

r1 @ X≤Y ⇔ X=Y | $true$.
r2 @ X≤Y ∧ Y≤X ⇔ X=Y.
r3 @ X≤Y ∧ Y≤Z ⇒ X≤Z.
r4 @ X≤Y ∧ X≤Y ⇔ X≤Y.

The CHR program implements reflexivity (r1), antisymmetry (r2), transitivity (r3) and idempotence (r4) in a straightforward way. The reflexivity rule r1 states that X≤Y is logically true, provided it is the case that X=Y. The antisymmetry rule r2 means X≤Y ∧ Y≤X is logically equivalent to X=Y. The transitivity rule r3 states that the conjunction of X≤Y and Y≤Z implies X≤Z.

2.2. *Operational Semantics of CHR*

The operational semantics of CHR is given by a transition system. A state is simply a goal, i.e. a conjunction of built-in and CHR constraints. Given a CHR program P we define the transition relation $\mapsto_P$ by introducing two computation steps (transitions), one for each kind of CHR rule (cf. Figure 1). In the figure, all meta-variables stand for conjunctions of constraints. The notation G_{bi} denotes the built-in constraints of G. Since the two transitions are structurally very similar, we first describe their common

Simplify

If	$(H \Leftrightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\,(G_{bi} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P^{\mathbf{Simplify}} (G \wedge B \wedge C \wedge H{=}H')$

Propagate

If	$(H \Rightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\,(G_{bi} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P^{\mathbf{Propagate}} (H' \wedge G \wedge B \wedge C \wedge H{=}H')$

Figure 1. *Computation Steps of Constraint Handling Rules*

behavior and only at the end point out their differences. A fresh variant of a rule is *applicable to a state* $H' \wedge G$ if H' matches its head H and if its guard C is implied by the built-in constraints appearing in G. A "fresh variant" of a rule is obtained by renaming its variables to fresh variables, listed in the sequence $\bar{x}$. "Matching" means that H' is an instance of H, i.e. it is only allowed to instantiate (bind) variables of H but not variables of H'. This is achieved by equating H' and H but existentially quantifying all variables from the rule, $\bar{x}$. This equation $H'{=}H$ is shorthand for pairwise equating the arguments of the constraints in H' and H, provided their constraint symbols are equal.

If an applicable rule is applied, the equation $H{=}H'$, its guard C and its body B are added to the resulting state. A rule application cannot be undone (CHR is a committed-choice language without backtracking). When a simplification rule is applied in the transition **Simplify**, the matching CHR constraints H' are removed from the state. The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints H' in the resulting state. Trivial non-termination caused by applying the same propagation rule again and again is avoided by applying it at most once to the same constraints [ABD 97].

A *computation* of a goal G in a program P is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto_P S_{i+1}$ beginning with the initial state S_0 for G and ending in a final state or diverging. A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent. $\mapsto_P^*$ denotes the reflexive and transitive closure of $\mapsto_P$. When it is clear from the context, we will drop the reference to the program P.

Example 2.2 Recall the solver program for $\leq$ of Example 2.1. Operationally the rule `r1` removes occurrences of constraints that match $\mathtt{X}\leq\mathtt{X}$. The antisymmetry rule `r2` means that if we find $\mathtt{X}\leq\mathtt{Y}$ as well as $\mathtt{Y}\leq\mathtt{X}$ in the current store, we can replace them by the logically equivalent `X=Y`. The transitivity rule `r3` propagates constraints. We add the logical consequence $\mathtt{X}\leq\mathtt{Z}$ as a redundant constraint. The idempotence rule `r4` absorbs multiple occurrences of the same constraint.

A computation of the goal $\mathtt{A}\leq\mathtt{B} \wedge \mathtt{C}\leq\mathtt{A} \wedge \mathtt{B}\leq\mathtt{C}$ proceeds as follows :

$$
\begin{array}{ll}
\underline{\mathtt{A}\leq\mathtt{B}} \wedge \underline{\mathtt{C}\leq\mathtt{A}} \wedge \mathtt{B}\leq\mathtt{C} & \mapsto^{\mathbf{Propagate}} \\
\mathtt{A}\leq\mathtt{B} \wedge \mathtt{C}\leq\mathtt{A} \wedge \underline{\mathtt{B}\leq\mathtt{C}} \wedge \underline{\mathtt{C}\leq\mathtt{B}} & \mapsto^{\mathbf{Simplify}} \\
\underline{\mathtt{A}\leq\mathtt{B}} \wedge \underline{\mathtt{B}\leq\mathtt{A}} \wedge \mathtt{B}=\mathtt{C} & \mapsto^{\mathbf{Simplify}} \\
\mathtt{A}=\mathtt{B} \wedge \mathtt{B}=\mathtt{C} &
\end{array}
$$

2.3. *Confluence*

The confluence property of a program guarantees that any computation for a goal results in the same final state no matter which of the applicable rules are applied.

Definition 2.2 A CHR program is called *confluent* if for all states S, S_1, S_2 : If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then S_1 and S_2 are joinable. Two states S_1 and S_2 are called *joinable* if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are variants[1].

To analyze confluence of a given CHR program we cannot check joinability starting from any given ancestor state S, because in general there are infinitely many such states. However for terminating programs, one can restrict the joinability test to a finite number of "minimal" states.

A CHR program is called *terminating*, if there are no infinite computations. For many existing CHR programs simple well-founded orderings are sufficient to prove termination [FRÜ 00]. In general, such orderings are not sufficient because of non-trivial interactions between simplification and propagation rules. In this paper we assume that the constraint solvers are terminating.

Definition 2.3 Let R_1 be a simplification rule and R_2 be a (not necessarily different) rule, whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head and C_i be the guard of rule R_i $(i = 1, 2)$. Then a *critical ancestor state of* R_1 *and* R_2 is

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

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1{=}A_2) \wedge C_1 \wedge C_2)$.

Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2 then the tuple (S_1, S_2) is a *critical pair* of R_1 and R_2. A critical pair (S_1, S_2) is *joinable*, if S_1 and S_2 are joinable.

1. Two states are variants of each other if they are equal up to a variable renaming.

The following theorem from [ABD 97, ABD 99b] gives a decidable, sufficient and necessary condition for confluence of a terminating CHR program :

Theorem 2.1 A terminating CHR program is confluent iff all its critical pairs are joinable.

Example 2.3 Recall the program for $\leq$ of Example 2.1. The following critical pair stems from the critical ancestor state $\mathtt{X}\leq\mathtt{Y} \wedge \mathtt{Y}\leq\mathtt{X} \wedge \mathtt{Y}\leq\mathtt{Z}$ of `r2` and `r3`

$$(S_1, S_2) = (\mathtt{X}{=}\mathtt{Y}\ \wedge\ \mathtt{X}\leq\mathtt{Z},\quad \mathtt{X}\leq\mathtt{Y}\ \wedge\ \mathtt{Y}\leq\mathtt{X}\ \wedge\ \mathtt{Y}\leq\mathtt{Z}\ \wedge\ \mathtt{X}\leq\mathtt{Z})$$

is joinable : S_1 is a final state, i.e. no computation step is possible. A computation beginning with S_2 results in S_1 :

$$\begin{array}{ll}
\underline{\mathtt{X}\leq\mathtt{Y}}\ \wedge\ \underline{\mathtt{Y}\leq\mathtt{X}}\ \wedge\ \mathtt{Y}\leq\mathtt{Z}\ \wedge\ \mathtt{X}\leq\mathtt{Z} & \mapsto^{\mathbf{Simplify}} \\
\underline{\mathtt{X}\leq\mathtt{Z}}\ \wedge\ \underline{\mathtt{X}\leq\mathtt{Z}}\ \wedge\ \mathtt{X}{=}\mathtt{Y} & \mapsto^{\mathbf{Simplify}} \\
\mathtt{X}\leq\mathtt{Z}\ \wedge\ \mathtt{X}{=}\mathtt{Y} &
\end{array}$$

2.4. *Completion*

Completion is the process of adding rules to a non-confluent program until it becomes confluent. Rules are built from a non-joinable critical pair to allow a transition from one of the states into the other while maintaining termination. In contrast to other completion methods, in CHR we need in general more than one rule to make a critical pair joinable : a simplification rule and a propagation rule [ABD 98]. When these rules are added, new critical pairs may be produced, but also old non-joinable critical pairs may be removed, because the new rules make them joinable. Completion tries to continue introducing rules this way until the program becomes confluent. The essential part of a completion algorithm is the introduction of rules from critical pairs.

Definition 2.4 Let $\gg$ be a termination order and let $(C_{ud1} \wedge C_{bi1}\ ,\ C_{ud2} \wedge C_{bi2})$ be a critical pair, where the states are ordered such that C_{ud1} is a non-empty conjunction and $C_{ud1} \gg C_{ud2}$. Then the *orientation* of the critical pair results in the two rules :

$$\begin{array}{ll}
C_{ud1} \Leftrightarrow C_{bi1} \mid C_{ud2} \wedge C_{bi2} & \\
C_{ud2} \Rightarrow C_{bi2} \mid C_{bi1} & \text{if } C_{ud2} \text{ is a non-empty conjunction and} \\
 & CT \not\models C_{bi2} \rightarrow C_{bi1}
\end{array}$$

Examples of completion will be shown in the next section of the paper. In these examples, unless otherwise noticed, a simple termination order will suffice, where $C_1 \gg C_2$ if $C_1 = (C_2 \wedge C)$, i.e. the conjunction C_1 contains all conjuncts of C_2 and more (C is non-empty).

In [ABD 98] it was shown that if the completion procedure stops successfully, then the resulting program is well-behaved. But completion cannot always be successful : completion is aborted if a critical pair cannot be transformed into rules. Completion may not terminate, because new rules produce new critical pairs.

2.5. *Operational Equivalence*

The following definition clarifies when two programs are operationally equivalent : if for each goal, all final states in one program are the same as the final states in the other program.

Definition 2.5 Let P_1 and P_2 be programs. A state S is P_1, P_2*-joinable*, iff there are two computations $S \mapsto^*_{P_1} S_1$ and $S \mapsto^*_{P_2} S_2$, where S_1 and S_2 are final states, and S_1 and S_2 are variants of each other.

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

In [ABD 99a], we gave a decidable, sufficient and necessary syntactic condition for operational equivalence of well-behaved CHR programs : when testing operational equivalence, similar to our confluence test, we can restrict ourselves to a finite number of *critical states* that consist of the head and the guard of a rule. These critical states are run in both programs, and their outcome must be the same.

Definition 2.6 Let P_1 and P_2 be programs. Then *a critical state of* P_1 *and* P_2 is defined as follows :

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

Theorem 2.2 Two well-behaved 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.

Examples for operational equivalence can be found in the subsequent sections.

3. Tight Integration of CHR Constraint Solvers with Completion

In the introduction, Example 1.1 illustrated that the union of two well-behaved (i.e. terminating and confluent) programs is not necessarily well-behaved. Once termination of the union has been established, we can use our confluence test to check if the union of well-behaved programs is confluent again. We call such programs “compatible”.

Definition 3.1 Let P_1 and P_2 be two well-behaved 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.

The critical pairs of $P_1 \cup P_2$ are the critical pairs of P_1 unioned with the critical pairs of P_2 unioned with critical pairs coming from one rule from P_1 and one rule from P_2. Since P_1 and P_2 are already confluent, for compatibility it suffices to check only those critical pairs coming from rules in different programs (cf. proof of upcoming Theorem 4.1). In other words, the confluence test can be made incremental in the addition of rules.

If the compatibility test succeeds, we can just take the union of the rules in the two programs. This holds even for constraints that are fully or partially defined in more than one of the programs which are merged.

Example 3.1 The well-behaved program P_1 contains the following CHR rules defining `max`, where `max(X,Y,Z)` means that `Z` is the maximum of `X` and `Y` :

$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X<Y | Z=Y.}$
$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X}\geq\texttt{Y | Z=X.}$

whereas well-behaved P_2 defines `max` by

$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y | Z=Y.}$
$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X>Y | Z=X.}$

Note that $<$, $\leq$, and $\geq$ are built-in constraints in this example.

In order to perform the union of the two programs, we check whether the definitions of `max` are compatible. There are three critical ancestor states coming from one rule in P_1 and one rule in P_2 :

$$\begin{array}{lllll} \texttt{max(X,Y,Z)} & \wedge & \texttt{X<Y} & \wedge & \texttt{X}\leq\texttt{Y} \\ \texttt{max(X,Y,Z)} & \wedge & \texttt{X}\geq\texttt{Y} & \wedge & \texttt{X}\leq\texttt{Y} \\ \texttt{max(X,Y,Z)} & \wedge & \texttt{X}\geq\texttt{Y} & \wedge & \texttt{X>Y} \end{array}$$

Since the critical pairs of these critical ancestor states are joinable, the two definitions of `max` are compatible. Hence we can just take the union of the rules and define `max` by all four rules.

Note that the constraint `max` is *"operationally stronger"* in $P_1 \cup P_2$ than in each program alone, in the sense that more computation steps are possible : in $P_1 \cup P_2$ (and P_1) we have the computation

$$\texttt{max(X,Y,Z)} \wedge \texttt{X}\geq\texttt{Y} \quad \mapsto_{P_1 \cup P_2} \quad \texttt{Z=X} \wedge \texttt{X}\geq\texttt{Y}$$

while in P_2 the goal cannot reduce at all, it is a final state. But like P_2, P_1 is not as strong as $P_1 \cup P_2$: the goal $\texttt{max(X,Y,Z)} \wedge \texttt{X}\leq\texttt{Y}$ is a final state in P_1, while it has a non-trivial computation in $P_1 \cup P_2$ and P_2.

Example 3.2 Here we consider a variation on a solver for `max` that does not use any built-in constraints (except for implicit syntactical equality). We define `max` with the inequalities as CHR constraints in two steps.

Given the constraint solver for $\leq$ (example 2.1), we add the following simplification rule describing the interaction of `max` and $\leq$:

$\texttt{max1 @ max(X,Y,Z)} \wedge \texttt{X}\leq\texttt{Y} \Leftrightarrow \texttt{Z=Y} \wedge \texttt{X}\leq\texttt{Y.}$

The resulting solver becomes non-confluent. The critical ancestor state `max(X,X,Z)` $\wedge$ $\texttt{X}\leq\texttt{X}$ of the rule `max1` and of the reflexivity rule `r1` of $\leq$ produces the non-joinable critical pair $(\texttt{X=Z} \wedge \texttt{X}\leq\texttt{X},\ \texttt{max(X,X,Z)})$. We use *completion* to make the solver confluent. For the above-mentioned critical pair it adds the rule :

$\texttt{max2 @ max(X,X,Z)} \Leftrightarrow \texttt{Z=X.}$

Now, we consider a solver for $<$ which is well-behaved

```
X<X ⇔ false.
X<Y ∧ X<Y  ⇔  X<Y.
X<Y ∧ Y<Z  ⇒  X<Z.
```

and we add the rule describing the interaction of `max` and $<$:

```
max3 @ max(X,Y,Z) ∧ Y<X  ⇔  Z=X ∧ Y<X.
```

The resulting solver remains well-behaved.

Finally, we union the solvers for $\leq$ and for $<$ that have been extended by the three rules for `max`, i.e. `max1`, `max2`, and `max3`. The union of these solvers is not confluent. The completion method adds the following rule to make a non-joinable critical pair stemming from the rules `max1` and `max3` joinable :

```
X≤Y ∧ Y<X  ⇔  false.
```

The rules derived by completion revealed interesting properties of `max` and the interaction of $\leq$ and $<$. The completed program is well-behaved.

4. Modularity of Termination and Confluence

We have seen that well-behavedness is not modular, i.e. it is not preserved under union of programs. We may ask ourselves if there are syntactic criteria for classes of programs that admit modularity of well-behavedness. In this section we will show that while for confluence, the answer is positive and simple (presupposing termination), the situation seems very difficult for termination.

When the two solvers do not have any *defined* CHR constraints in common, we call them *non-overlapping*. Note that non-overlapping solvers may have common (shared) CHR constraints and function symbols and have common built-in constraints (by definition, at least syntactical equality). We can show that the union of two non-overlapping well-behaved solvers is always well-behaved if the union is terminating.

Theorem 4.1 Let P_1 and P_2 be two well-behaved CHR programs and let the union of the two programs, $P_1 \cup P_2$, be terminating. If P_1 and P_2 are non-overlapping then $P_1 \cup P_2$ is confluent.

Proof. To show that $P_1 \cup P_2$ is confluent, we only have to show that all critical pairs of $P_1 \cup P_2$ are joinable, since $P_1 \cup P_2$ is terminating. The set of critical pairs of $P_1 \cup P_2$ consists of all critical pairs stemming from two rules appearing in P_1 (case 1. below), all critical pairs stemming from two rules appearing in P_2 (case 2) and all critical pairs stemming from one rule appearing in P_1 and one rule appearing in P_2 (case 3).

1) P_1 is well-behaved, thus all critical pairs stemming from two rules appearing in P_1 are joinable. Therefore, these critical pairs are also joinable in $P_1 \cup P_2$.

2) Analogous to case 1.

3) Critical pairs from rules of different programs can only exist, if the head of the rules have at least one constraint in common. Since P_1 and P_2 are non-overlapping, there exists no critical pair stemming from one rule in P_1 and one rule in P_2. □

For modularity of termination, the situation seems very difficult : even if two terminating programs do not have common CHR constraint symbols, their union may be non-terminating.

Example 4.1 Consider the following two programs :

$P1$: `c(f(X))` $\Leftrightarrow$ `X=g(Y)` $\wedge$ `c(Y).`
$P2$: `d(g(Y))` $\Leftrightarrow$ `Y=f(Z)` $\wedge$ `d(Z).`

Any goal (of finite size) terminates in each of the two programs, but the goal `c(f(X))` $\wedge$ `d(X)` does not terminate in the union of the programs (due to common function symbols).

`c(f(X))` $\wedge$ `d(X)` $\mapsto^{\textbf{Simplify}}$
`X=g(Y)` $\wedge$ `c(Y)` $\wedge$ `d(g(Y))` $\mapsto^{\textbf{Simplify}}$
`X=g(f(W))` $\wedge$ `Y=f(W)` $\wedge$ `c(f(W))` $\wedge$ `d(W)` $\mapsto^{\textbf{Simplify}}$ ...

Actually, even if there are no common symbols in the program text, we may run into trouble.

Example 4.2 The previous example can be rewritten such that instead of common function symbols one uses built-in constraints to the same effect :

$P1$: `c(FX)` $\Leftrightarrow$ `f1(FX,X) | g1(X,Y)` $\wedge$ `c(Y).`
$P2$: `d(GY)` $\Leftrightarrow$ `g2(GY,Y) | f2(Y,Z)` $\wedge$ `d(Z).`

where $f1(X,Y)$ and $f2(X,Y)$ are both defined as $X = f(Y)$ in the constraint theory for the built-in constraints and analogously for $g1(X,Y)$ and $g2(X,Y)$. There are no common symbols in the CHR program itself, but only in the constraint theory. Any goal terminates in each of the two programs, but the goal `c(FX)` $\wedge$ `f1(FX,X)` $\wedge$ `d(X)` does not terminate in the union of the programs.

Summarizing, as soon as there are common symbols, no matter if they are CHR constraints, built-in constraints or function symbols (even when only shared in the built-in constraint theories), termination is in danger. But any non-trivial integration of constraint solvers will at least share some function symbols, otherwise there could not be shared variables in goals, and without shared variables there is no non-trivial communication between the solvers.

5. Cooperation Using Bridge Rules and Completion

In practice, one will often add to the union of non-overlapping solvers a few so-called *bridge rules*. These are rules that may translate constraints from one solver to constraints of the other solver to improve the overall solving power. In general, they relate constraints from different solvers to enable non-trivial cooperation. In other words, they define *communication* between the solvers by sharing data (constraints).

When adding bridge rules, care has to be taken to maintain termination. On the other hand, bridge rules can be used to re-introduce termination : we may make a union of solvers terminating by renaming symbols apart and using bridge rules to control the interaction between the solvers. In any case, terminating bridge rules will typically cause non-confluence and thus will be the starting point for completion.

Example 5.1 We want to build a Boolean constraint solver from a well-behaved program P_1 defining conjunction and a well-behaved program P_2 defining implication. In P_1, the constraint `and(X,Y,Z)` stands for `X` $\wedge$ `Y` $\leftrightarrow$ `Z` and in P_2, `imp(X,Y)` stands for `X` $\rightarrow$ `Y`.

```
P1: and(X,X,Z) ⇔ X=Z.
    and(X,Y,1) ⇔ X=1 ∧ Y=1.
    and(X,1,Z) ⇔ X=Z.
    and(X,0,Z) ⇔ Z=0.
    and(1,Y,Z) ⇔ Y=Z.
    and(0,Y,Z) ⇔ Z=0.
    and(X,Y,Z) ∧ and(X,Y,Z1) ⇔ and(X,Y,Z) ∧ Z=Z1.

P2: imp(0,X) ⇔ true.
    imp(X,0) ⇔ X=0.
    imp(1,X) ⇔ X=1.
    imp(X,1) ⇔ true.
    imp(X,Y) ∧ imp(Y,X) ⇔ X=Y.
```

We add the following bridge rule :

```
and(X,Y,X) ⇔ imp(X,Y).
```

The program containing P_1 and P_2 together with the bridge rule is not confluent : the critical pair (*true*, `imp(X,X)`) stemming from the critical ancestor state `and(X,X,X)` of the first rule of `and` and the bridge rule is not joinable. Completion generates the following rules from the non-joinable critical pairs :

```
imp(X,X) ⇔ true.
imp(X,Y) ∧ imp(X,Y) ⇔ imp(X,Y).
imp(X,Y) ∧ and(X,Y,Z) ⇔ imp(X,Y) ∧ X=Z.
```

Again, the automatically derived rules reveal interesting properties of the constraints.

6. Removal of Redundant Rules with Operational Equivalence

We can use a variation of the operational equivalence test [ABD 99a] between programs to remove redundant rules from the (completed) union of constraint solvers.

Definition 6.1 A rule R is redundant in a CHR program P iff for all states S : If $S \mapsto_P^* S_1$ then $S \mapsto_{P\setminus\{R\}}^* S_2$, where S_1 and S_2 are final states and S_1 and S_2 are variants of each other.

Example 6.1 In example 3.1, the union of the two programs defining `max`

```
r1 @ max(X,Y,Z) ⇔ X<Y | Z=Y.
r2 @ max(X,Y,Z) ⇔ X≥Y | Z=X.
r3 @ max(X,Y,Z) ⇔ X≤Y | Z=Y.
r4 @ max(X,Y,Z) ⇔ X>Y | Z=X.
```

was operationally stronger than each program alone. However, the union contains redundant rules. For example, rule `r3` can always make a transition when rule `r1` does, with the same result, but not vice versa. Hence rule `r1` is redundant, and analogously for rule `r4`.

Redundant rules can be discovered using operational equivalence : We remove one rule from the program and compare it with the original program. If the two programs are operationally equivalent, then the rule was obviously redundant and we can remove it. We continue until we have tried to remove all rules. The final program found this way is not necessarily unique, since the result may depend on the order in which rules are tried and removed.

However, Theorem 2.2 may not be applicable for our redundancy check : If we remove a rule from a well-behaved program, it may become non-confluent. In order to come up with a decidable rule redundancy test, we first have to test confluence of the program without the candidate rule for redundancy. If the program is not confluent, it cannot be operationally equivalent to the initial program, and hence the candidate rule cannot be redundant. If the program is confluent, we can and must check for operational equivalence.

Theorem 6.1 Let P be a well-behaved program. A rule R is redundant with respect to P iff $P\backslash\{R\}$ is well-behaved and P and $P\backslash\{R\}$ are operationally equivalent.

Proof. First, we prove the claim that $P\backslash\{R\}$ is well-behaved by contradiction. Assumption : $P\backslash\{R\}$ is not well-behaved. We can distinguish two cases :

1) $P\backslash\{R\}$ is non-terminating, thus P is also non-terminating, which is a contradiction to the fact that P is well-behaved.

2) $P\backslash\{R\}$ is non-confluent, thus there exists a state S such that $S \mapsto^*_{P\backslash\{R\}} S_1$ and then $S \mapsto^*_{P\backslash\{R\}} S_2$, where S_1 and S_2 are final states, and S_1 and S_2 are not variants of each other. R is redundant with respect to P, therefore there exists a state S_3 such that $S \mapsto^*_P S_3$, where S_3 is a final state, and S_3, S_1 as well as S_3, S_2 are variants of each other. This is a contradiction to the claim that S_1 and S_2 are not variants of each other.

Now we prove that P and $P\backslash\{R\}$ are operationally equivalent. R is redundant with respect to P, thus for all states S the following holds : $S \mapsto^*_P S_1$ then $S \mapsto^*_{P\backslash\{R\}} S_2$, where S_1 and S_2 are final states and S_1 and S_2 are variants of each other. Therefore, all states are $P, P\backslash\{R\}$-joinable. □

It is easy to see that we can specialize our operational equivalence test for redundancy removal : We only have to check if the computation step due to the candidate rule that is tested for redundancy can be performed by the remainder of the program, but we do not have to consider any other rule prefixes.

Example 6.2 The critical states of the program P in Example 6.1 are

```
cs1: max(X,Y,Z) ∧ X<Y
cs2: max(X,Y,Z) ∧ X≥Y
cs3: max(X,Y,Z) ∧ X≤Y
cs4: max(X,Y,Z) ∧ X>Y
```

Note that any subset of the program in Example 6.1 is still well-behaved. A program $P \backslash \{R\}$ ($R \in \{\mathtt{r1}, \mathtt{r2}, \mathtt{r3}, \mathtt{r4}\}$) obviously cannot contribute any new critical states. So if we try to remove rule `r1` we only have to check the critical state from rule `r1`, that is `cs1`, by running it in both programs :

$\mathtt{max(X,Y,Z)} \wedge \mathtt{X{<}Y} \mapsto_P \mathtt{X{<}Y} \wedge \mathtt{Z{=}Y}$ by rule `r1`
$\mathtt{max(X,Y,Z)} \wedge \mathtt{X{<}Y} \mapsto_{P\backslash\{\mathtt{r1}\}} \mathtt{X{<}Y} \wedge \mathtt{Z{=}Y}$ by rule `r3`

Since rule `r3` enables the same transition, rule `r1` must be redundant. In an analogous way, redundancy of rule `r4` can be shown. Rule `r2`, however, is not redundant :

$\mathtt{max(X,Y,Z)} \wedge \mathtt{X{\geq}Y} \mapsto_P \mathtt{X{\geq}Y} \wedge \mathtt{Z{=}X}$ by rule `r2`
$\mathtt{max(X,Y,Z)} \wedge \mathtt{X{\geq}Y} \not\mapsto_{P\backslash\{\mathtt{r2}\}}$

In program $P \backslash \{\mathtt{r2}\}$, the critical state is a final state. Hence (the only) redundancy free program consists of the rules `r2` and `r3`.

7. Conclusions

In this paper, we have shown that terminating and confluent, i.e. well-behaved CHR constraint solvers can be merged provided termination is preserved : their tight integration is the union of the rules, even if some constraints are fully or partially defined and/or used in several solvers or program parts. In case that the resulting solver becomes non-confluent, we use our completion method to improve its behavior.

Non-overlapping solvers do not define common constraints but may freely share them otherwise. We have shown that their union is always well-behaved if it is terminating. We argued that a similar modularity result for termination is likely to be very hard to obtain. Future work will investigate how to maintain termination of the union, i.e. modularity results, trying to build on work in term rewriting systems such as [OHL 95].

We have discussed bridge rules as a communication means to integrate solvers with disjoint constraints utilizing completion. Finally, we have presented a method to remove redundant rules from a CHR solver using our operational equivalence test and our confluence test.

For future work, we are also interested in general notions of confluence and completion, since we have found that on larger examples, their current requirements are unnecessarily strict. Finally, a more efficient method for detecting and removing redundant rules should be found.

8. Bibliographie

[ABD 97] ABDENNADHER S., « Operational Semantics and Confluence of Constraint Propagation Rules », *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330, Springer-Verlag, 1997.

[ABD 98] ABDENNADHER S., FRÜHWIRTH T., « On Completion of Constraint Handling Rules », *4th International Conference on Principles and Practice of Constraint Programming, CP98*, LNCS 1520, Springer-Verlag, 1998.

[ABD 99a] ABDENNADHER S., FRÜHWIRTH T., « Operational Equivalence of Constraint Handling Rules », *Fifth International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS, Springer-Verlag, 1999.

[ABD 99b] ABDENNADHER S., FRÜHWIRTH T., MEUSS H., « Confluence and Semantics of Constraint Simplification Rules », *Constraints Journal*, vol. 4, n° 2, 1999.

[BAA 01] BAADER F., SCHULZ K. U., « Combining Constraint Solving », *Constraints in Computational Logics*, 2001.

[CAS 00] CASTRO C., MONFROY E., « Basic Operators for Solving Constraints via Collaboration of Solvers », *Proceedings of AISC 2000*, LNAI 1930, Springer-Verlag, 2000.

[FRÜ 98] FRÜHWIRTH T., *A Declarative Language for Constraint Systems : Theory and Practice of Constraint Handling Rules*, Habilitation, Computer Science Institute, LMU Munich, 1998.

[FRÜ 00] FRÜHWIRTH T., « Proving Termination of Constraint Solver Programs », K.R. APT A.C. KAKAS E. M., ROSSI F., Eds., *New Trends in Constraints*, LNAI 1865, Springer-Verlag, 2000.

[GRA 01] GRANVILLIERS L., MONFROY E., BENHAMOU F., « Cooperative Solvers in Constraint Programming : A Short Introduction », *Workshop on Cooperative Solvers in Constraint Programming (CoSolv) at CP 2001*, Paphos, Cyprus, December 2001.

[HOF 00] HOFSTEDT P., « Better Communication for Tighter Cooperation », *First Intl. Conference on Computational Logic (CL 2000)*, LNAI 1861, Springer-Verlag, 2000.

[MON 00] MONFROY E., « The Constraint Solver Collaboration Language of BALI », *Frontiers of Combining Systems 2*, Vol. 7 of Studies in Logic and Computation, Research Studies Press/Wiley, 2000.

[OHL 95] OHLEBUSCH E., « Modular Properties of Composable Term Rewriting Systems », *Journal of Symbolic Computation*, vol. 20, n° 1, 1995.

[TIN 98] TINELLI C., RINGEISSEN C., « Non-Disjoint Unions of Theories and Combinations of Satisfiability Procedures : First Results », rapport, avril 1998, Department of Computer Science, University of Illinois at Urbana-Champaign.