Aanvraag specialisatiebeurs IWT

Beschrijving van het onderzoeksproject

"Analyse en Optimalisatie van CHR-programma's"

Jon Sneyers

September 2005

Inhoudsopgave

1 Probleemstelling

Constraint Logic Programming [37, 45] (CLP) combineert de voordelen van logisch programmeren en *constraint* programmeren. In *logische programmeertalen* worden problemen op een declaratieve manier geformuleerd in de vorm van regels die bepaalde relaties (predicaten) definiëren. Problemen worden opgelost door een ingebouwd *backtracking*-mechanisme dat de verschillende keuzemogelijkheden doorzoekt. Bij *constraint* programmeren worden efficiënte probleem-specifieke algoritmes gebruikt om deelproblemen op te lossen waarin relaties voorkomen die beperkingen (*constraints*) uitdrukken. Een *constraint solver* is een inferentie-systeem dat basisoperaties op constraints ondersteunt.

In de begindagen van CLP werd gebruik gemaakt van de "black-box"-aanpak: efficiënte ingebouwde constraint solvers, geïmplementeerd in een low-level programmeertaal, die erg moeilijk aan te passen waren, laat staan te debuggen of te analyseren. Verscheidene "glass-box"-technieken zijn sindsdien voorgesteld [15, 22, 40] om constraint systemen beter aanpasbaar te maken, maar die technieken lieten nog niet toe om op een praktische manier rechtstreeks een solver te implementeren over een nieuw constraint domein.

Constraint Handling Rules [26, 27] (CHR) is een declaratieve high-level taal-uitbreiding die specifiek ontworpen is om constraint systemen op een meer flexibele ("no-box") en toepassingsgerichte manier te schrijven en aan te passen. Gebruiker-gedefinieerde constraints kunnen toegevoegd worden aan een gegeven *host*-taal. Omdat CHR een taal-*uitbreiding* is, heeft deze enkel betrekking op constraints en worden alle hulp-berekeningen rechtstreeks uitgevoerd in de *host*-taal. CHR wordt gebruikt in tientallen projecten en er bestaan implementaties van CHR in onder andere **Java** [3, 73, 75] en **Prolog** [43, 56] (zie sectie 3.2.1 voor een bondig overzicht). CHR en uitbreidingen ervan [2, 38] zijn nuttig gebleken voor de implementatie van redeneersystemen in het algemeen [44].

Het belangrijkste nadeel van huidige CHR-implementaties ten opzichte van low-level implementaties van specifieke constraint solvers is de slechtere performantie. Verschillende optimalisaties zijn mogelijk, zij het via declaraties die door de gebruiker (programmeur) gespecifieerd worden of door (automatische) statische of dynamische analyse. Eigenschappen die essentieel zijn voor de analyse van CHR-programma's zijn: operationele *semantiek* [1, 24, 27] en *equivalentie* [7], *confluentie* [4, 5, 6], *terminatie* [28], *complexiteit* [29, 30, 69], en *implicatie* [57]. Verder onderzoek naar deze en andere eigenschappen is nodig. Om de performantie van CHR-programma's te verbeteren kunnen bestaande optimalisatietechnieken voor CHR-programma's [23, 31, 42, 54] worden verfijnd en veralgemeend, en kunnen nieuwe optimalisaties [66] worden ontwikkeld en geïmplementeerd.

CHR was oorspronkelijk ontworpen voor het ontwikkelen van toepassing-specifieke constraint solvers, maar wordt steeds meer gebruikt als een *general-purpose* programmeertaal. Recent onderzoek [32, 58] geeft aan dat verschillende klassieke algoritmes elegant en compact geïmplementeerd kunnen worden in CHR, met optimale complexiteit, dankzij bestaande optimalisaties zoals *hash-table constraint stores*. Inderdaad, in [69] toonden we aan dat elk Turing of RAM-machine programma geïmplementeerd kan worden in CHR met beste tijds- en ruimtecomplexiteit. Ondanks dit theoretisch resultaat is het voor veel klassieke algoritmes nog een open probleem of de bestaande optimalisaties een natuurlijke en elegante CHR implementatie toelaten.

2 Doelstelling

De belangrijkste doelstelling van dit project is het ontwerpen, implementeren, evalueren en publiceren van nieuwe analyses en optimalisaties voor CHR-programma's, alsmede het verbeteren en veralgemenen van bestaande analyses en optimalisaties. Op die manier kunnen we de performantie – en dus de praktische bruikbaarheid – verbeteren van bestaande CHR-implementaties, waarbij we ons vooral zullen richten op het K.U.Leuven CHR systeem. Ook zullen we heel wat aandacht besteden aan de *language design* van CHR, meer bepaald kunnen we nieuwe syntactische elementen voorstellen zoals type-, mode- en andere declaraties, syntactische suiker, modules, compilerdirectieven, ... De bijkomende declaraties kunnen dan gebruikt worden om de optimalisaties te versterken. Ook kunnen we (onderdelen van) een gebruiksvriendelijke *environment* ontwikkelen, waarin veelvoorkomende programmeerfouten gedetecteerd worden (o.a. op basis van type-checking) tijdens het schrijven van CHR-programma's. Op die manier wordt het ontwikkelen van correcte CHR-programma's (nog) gemakkelijker.

In het algemeen is onze doelstelling dus het verhogen van de bruikbaarheid van CHR, enerzijds op het gebied van performantie, anderzijds op het gebied van gebruiksvriendelijkheid voor de CHR programmeur. De analyses die in de volgende sectie besproken worden zullen we enerzijds gebruiken als bouwstenen voor verschillende hulpmiddelen voor de programmeur in een gebruiksvriendelijke *environment*, en anderzijds als basis voor verschillende optimalisaties in de CHR compiler.

Een belangrijk middel om onze doelstelling te bereiken is het implementeren in CHR van klassieke en recente algoritmes, en vervolgens die implementatie analyseren (zoals in [58]) om eventuele *bottlenecks* te identificeren, wat vervolgens kan leiden tot verdere verbeteringen in de CHR compiler en het run-time systeem.

Onze onderzoekseenheid heeft in het verleden reeds vele belangrijke bijdragen geleverd in de domeinen van analyse van (constraint) logische programmeertalen [12, 13, 18] (o.a. *abstract interpretation* [9, 10, 11, 14, 46, 47]), en toepassingen van CLP [21, 48, 49, 50]. Ook specifiek binnen het relatief recente domein van Constraint Handling Rules is er binnen onze onderzoekseenheid een behoorlijke expertise [54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65, 66, 67, 68, 69, 73]. In het afgelopen jaar werd er in onze onderzoekseenheid een doctoraat behaald dat integraal over CHR handelde [53]. Het K.U.Leuven CHR-systeem [56] is een state-of-the-art implementatie van CHR die momenteel voor drie **Prolog**-omgevingen beschikbaar is: **hProlog** [19], **SWI-Prolog** [63, 74] en **XSB** [61, 62, 72]. Dit systeem wordt ontwikkeld door Tom Schrijvers, Bart Demoen, en mijzelf, die deel uitmaken van de onderzoekseenheid waarbinnen dit project zal uitgevoerd worden: de subgroep *"Ontwerp, Analyse en Implementatie van Declaratieve Programmeertalen"* binnen de onderzoeksgroep *"Declaratieve Talen en Artificiële Intelligentie"* (DTAI) aan het departement Computerwetenschappen van de K.U.Leuven.

Onze onderzoekseenheid onderhoudt tevens contacten met buitenlandse onderzoekers, zoals de onderzoeksgroep van prof. Peter J. Stuckey [60] aan de universiteit van Melbourne, Australië; de onderzoeksgroep van prof. Kim Marriott [20] aan de universiteit van Monash, Australië; de onderzoeksgroep van prof. David S. Warren [61] aan de universiteit van Stony Brook, New York, USA; de onderzoeksgroep van prof. Thom Frühwirth [58], de oorspronkelijke ontwikkelaar van CHR, aan de universiteit van Ulm, Duitsland.

3 Projectbeschrijving

In sectie 3.1 leggen we bondig de syntax en semantiek van CHR uit. Sectie 3.2 bestaat uit een opsomming van bestaande CHR systemen en een schetsmatige beschrijving van hun werking en enkele eigen bijdragen aan de verbetering van het K.U.Leuven CHR systeem. Vervolgens, in sectie 3.3, geven we een overzicht van bestaande analyses en mogelijke verbeteringen en uitbreidingen. Tenslotte bespreken we in sectie 3.4 allerlei optimalisaties die gebaseerd zijn op deze analyses.

3.1 Inleiding: Constraint Handling Rules

In deze eerste sectie geven we een bondige introductie tot CHR. We beginnen met een eenvoudig voorbeeld van een CHR-programma. Vervolgens introduceren we de CHR-syntax. Tenslotte geven we de declaratieve en operationele semantiek. Voor een uitgebreider overzicht van CHR verwijzen we naar [24, 27].

3.1.1 Voorbeeld

We tonen een gebruiker-gedefinieerde constraint voor “kleiner-dan-of-gelijk-aan” (`=<`) die met variabele argumenten kan werken:

```
reflexiviteit  @ X=<Y  <=> X=Y | true.
antisymmetrie  @ X=<Y, Y=<X  <=> X=Y.
transitiviteit @ X=<Y, Y=<Z  ==> X=<Z.
```

Dit CHR-programma specifieert hoe de constraint `=<` vereenvoudigd en gepropageerd kan worden. In elke regel is het eerste deel commentaar (tot en met de `@`), een optionele string die genegeerd wordt door de compiler en gebruikt kan worden om CHR regels een naam te geven. De eerste regel, `reflexiviteit`, zegt dat `X=<Y` logisch waar is indien `X=Y`. Die conditie vormt de (optionele) *guard* van een regel, die voldaan moet zijn vooraleer de regel kan toegepast worden. Uit deze eerste regel volgt dat de constraint `X=<X` vereenvoudigd kan worden tot `true`. De tweede regel, `antisymmetrie`, betekent dat indien zowel de constraint `X=<Y` als `Y=<X` aanwezig zijn, we deze kunnen vervangen door de logisch equivalente (ingebouwde) constraint `X=Y`.

Deze twee regels zijn simplificatieregels (*simplification rules*), waarbij constraints vervangen worden door eenvoudigere – maar logisch *equivalente* – constraints. De derde regel, `transitiviteit`, drukt uit dat de constraints `X=<Y` en `Y=<Z` impliceren dat `X=<Z`. Een (overbodige) constraint die een logisch *gevolg* is van de oorspronkelijke constraints wordt toegevoegd. Dit soort regels noemen we propagatieregels (*propagation rules*).

De redundante constraints veroorzaakt door propagatieregels zijn nuttig. Beschouw bijvoorbeeld de query `A=<B,C=<A,B=<C`. In eerste instantie kunnen geen van beide simplificatieregels toegepast worden. De eerste twee constraints (`C=<A,A=<B`) zorgen er echter voor dat de `transitiviteit`-regel wordt toegepast zodat `C=<B` toegevoegd wordt aan de query. Samen met `B=<C` kan nu `antisymmetrie` worden toegepast zodat `B=<C` en `C=<B` vervangen worden door `B=C` en de oorspronkelijke query vereenvoudigd is tot `A=<B,C=<A,B=C`. Na substitutie kan `antisymmetrie` nogmaals toegepast worden, dit keer op `A=<B,B=<A`. Nu is de query opgelost, met als resultaat `A=B,B=C`.

3.1.2 Syntax

Een *constraint* is een speciaal eerste-orde predicaat. We gebruiken twee disjuncte verzamelingen van predicatensymbolen voor de twee verschillende klassen van constraints: enerzijds de voorgedefinieerde ingebouwde (*built-in*) constraints, anderzijds de gebruikergedefinieerde CHR constraints. De ingebouwde constraints worden behandeld door een bestaande solver uit de *host*-taal waaraan de taal-uitbreiding CHR werd toegevoegd (bijvoorbeeld **Prolog**). We beschouwen hier elk statement uit de *host*-taal dat voorkomt in het CHR-programma als een ingebouwde constraint.

Definitie 3.1 (CHR-programma) *Een CHR-programma is een eindige verzameling van CHR-regels. Er zijn drie soorten CHR-regels:*

- Simplificatieregels (*simplification* rules) zijn van de vorm

  $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j$ `|` $B_1, \ldots, B_k$

- Propagatieregels (*propagation* rules) zijn van de vorm

  $H_1, \ldots, H_i$ `==>` $G_1, \ldots, G_j$ `|` $B_1, \ldots, B_k$

- Simpagatieregels (*simpagation* rules) [1] zijn van de vorm

  $H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i$ `<=>` $G_1, \ldots, G_j$ `|` $B_1, \ldots, B_k$

met $i > 0$, $j \geq 0$, $k \geq 0$, $0 < l < i$ en waarbij de *multi-head* $H_1, \ldots, H_i$ een niet lege rij van CHR constraints voorstelt, de *guard* $G_1, \ldots, G_j$ een rij van ingebouwde constraints en de *body* $B_1, \ldots, B_k$ een rij van constraints (ingebouwd of CHR). Lege rijen worden voorgesteld door de ingebouwde constraint `true`. Voor de eenvoud kan een lege *guard* weggelaten worden samen met het "`|`"-symbool.

3.1.3 Declaratieve semantiek

In tegenstelling tot algemene *committed-choice* programma's kan aan CHR-programma's een *declaratieve semantiek* gegeven worden, aangezien ze enkel constraints definiëren en niet procedures in het algemeen. De declaratieve interpretatie van een CHR-programma P wordt gegeven door een conjunctie van universeel gequantificeerde logische formules $\mathcal{P}$, één voor elke regel, en een consistente ingebouwde *constraint theorie CT* die de betekenis van de ingebouwde constraints die voorkomen in het programma vastlegt. We veronderstellen dat CT een gelijkheid constraint `=` bevat en de basisconstraints `true` en `false`.

Met $\bar{x}$ noteren we de reeks (globale) variabelen die voorkomen in de *head* $H_1, \ldots, H_i$ van een regel. Met $\bar{y}$ ($\bar{z}$) noteren we de andere (lokale) variabelen die voorkomen in de *guard* $G_1, \ldots, G_j$ (*body* $B_1, \ldots, B_k$) van een regel maar niet in de *head*.

Definitie 3.2 (Declaratieve semantiek van CHR-regels) *De declaratieve interpretatie (logische betekenis) van een simplificatieregel is een logische* equivalentie *indien aan de* guard *voldaan is:*

[1] In wat volgt laten we simpagatieregels soms buiten beschouwing. In de declaratieve semantiek kunnen deze immers altijd geschreven worden als simplificatieregels, waarbij het gedeelte voor de backslash ($H_1, \ldots, H_l$) herhaald wordt in de *body*. Deze omzetting kan echter gevolgen hebben voor terminatie en efficiëntie.

$$\forall \bar{x} \forall \bar{y}((G_1 \wedge \ldots \wedge G_j) \rightarrow (H_1 \wedge \ldots \wedge H_i \leftrightarrow \exists \bar{z}(B_1 \wedge \ldots \wedge B_k)))$$

Een propagatieregel heeft als declaratieve interpretatie een logische implicatie *indien aan de* guard *voldaan is:*

$$\forall \bar{x} \forall \bar{y}((G_1 \wedge \ldots \wedge G_j) \rightarrow (H_1 \wedge \ldots \wedge H_i \rightarrow \exists \bar{z}(B_1 \wedge \ldots \wedge B_k)))$$

De declaratieve interpretatie van het CHR-programma uit 3.1.1 bestaat uit de ingebouwde *constraint theorie CT* samen met de conjunctie van volgende drie formules:

$$\begin{cases} \forall X, Y((X = Y) \rightarrow (X{=}{<}Y \leftrightarrow \texttt{true})) \\ \forall X, Y(X{=}{<}Y \wedge Y{=}{<}X \leftrightarrow X = Y) \\ \forall X, Y(X{=}{<}Y \wedge Y{=}{<}Z \rightarrow X{=}{<}Z) \end{cases}$$

3.1.4 Operationele semantiek

De (abstracte) operationele semantiek ω_t [27] van CHR-programma's wordt gegeven door een transitie-systeem waarin de toestand $\langle F, E, D \rangle$ uit drie conjuncties van constraints bestaat: de *query* (of *goal*) constraints F die nog moeten opgelost worden, de CHR-constraints E en de ingebouwde constraints D die reeds geaccumuleerd en vereenvoudigd werden. Samengevat (en enigszins vereenvoudigd) worden volgende vier stappen (transities, reducties) in willekeurige volgorde toegepast bij de uitvoering van een CHR-programma:

- In de *Solve*-stap (*Introduce*-stap) wordt een ingebouwde constraint (CHR-constraint) uit de query F opgelost (geïntroduceerd) zodat D (E) aangepast wordt.
- In de *Simplify*-stap wordt een simplificatieregel waarvan de *head* constraints overeenkomen met constraints uit E toegepast als de *guard* voldaan is; de *body* wordt toegevoegd aan F, de *head* wordt verwijderd uit E en de matching-substitutie wordt aan D toegevoegd
- In de *Propagate*-stap wordt een propagatieregel waarvan de *head* constraints overeenkomen met constraints uit E toegepast als de *guard* voldaan is; de *body* wordt toegevoegd aan F en de substitutie voor de matching wordt aan D toegevoegd (de *head* blijft in E)

De initiële toestand voor een query F is $\langle F, \texttt{true}, \texttt{true} \rangle$. Een gefaalde eindtoestand is van de vorm $\langle F, E, \texttt{false} \rangle$, een succesvolle eindtoestand is van de vorm $\langle \texttt{true}, E, D \rangle$ en $E \wedge D$ vormt een antwoord voor de query. Met de notatie $S_n \mapsto S_{n+1}$ wordt bedoeld dat er een transitiestap is die de toestand S_n reduceert tot S_{n+1}. Als S via verschillende transities naar S' te reduceren is noteren we $S \mapsto^{\star} S'$

De abstracte operationele semantiek is niet-deterministisch in die zin dat sommige aspecten van de uitvoeringsstrategie niet vastgelegd zijn. In concrete implementaties van CHR compilers ligt de uitvoeringsstrategie (meer gedetailleerd) vast, waardoor ze in feite een specifieke *instantiatie* van de abstracte semantiek implementeren. Voor veel CHR programma's volstaat de abstracte semantiek om eigenschappen als correctheid, confluentie en volledigheid aan te tonen. Om de correctheid van bepaalde optimalisaties en om sterkere terminatie- en complexiteitseigenschappen te garanderen is een meer gedetailleerde operationele semantiek noodzakelijk.

Verfijnde semantiek. Bijna alle huidige implementaties van CHR volgen een striktere, *verfijnde* semantiek ω_r (*refined operational semantics*) [24] waarin de regels niet in willekeurige volgorde, maar enkel in de volgorde zoals ze in het programma voorkomen, kunnen worden toegepast. Dit is van belang voor CHR-compilers aangezien dit sterkere optimalisaties kan toelaten. CHR-programma's vertonen tevens een beter terminatiegedrag in de verfijnde semantiek.

Oproep-gebaseerde verfijnde semantiek. Een conceptueel nog dichter bij de implementatiedetails aanleunende semantiek is de *oproep-gebaseerde* verfijnde semantiek ω_c (*call-based refined operational semantics*) [60]. In deze semantiek komt het proceduraal karakter van constraint-oproepen tot uiting. In [60] wordt aangetoond dat ω_c en ω_r equivalente operationele semantieken zijn.

Oproep-gebaseerde verfijnde semantiek voor *occurrence*-representaties. De vorige semantieken werden gedefinieerd op het niveau van CHR regels. De compilatie van CHR-regels naar *host*-taal code gebeurt echter niet per regel, maar per *occurrence* (voorkomen van constraint in het linkerlid van een regel). In [66] hebben we een mapping naar een intermediaire representatie (*occurrence representation*) voorgesteld, en een operationele semantiek ω_o die voor die mapping equivalent is met de vorige twee. Dit formeel kader wordt in [66] gebruikt om de correctheid te bewijzen van een nieuwe optimalisatie, die niet gedefinieerd kan worden als een source-to-source transformatie op CHR programma's maar wel als een transformatie op occurrence representations.

3.2 CHR compilers

3.2.1 Bestaande implementaties

De eerste CHR-implementatie, in de zomer van 1991, was een interpreter geschreven in **ECL^{i}PS^e Prolog** genaamd **Cheer** [25, 35]. Sindsdien zijn er implementaties van CHR in verschillende *host*-talen: **Common LISP** [41], **ECL^{i}PS^e** [34], **SICStus Prolog** [43] (de referentie-implementatie), **YAP Prolog** [17], **Java** [3, 75, 73], **HAL** [20, 42, 23], **Haskell** [70], **XSB** [61, 62, 56], **hProlog** [56] en **SWI-Prolog** [63, 56]. De K.U.Leuven CHR systemen voor **Prolog** [56] en **Java** [73] zijn *state-of-the-art* implementaties.

3.2.2 Werking

De CHR implementaties zijn zowat allemaal gebaseerd op compilatie naar de *host*-taal volgens algemene compilatieschema's, aangevuld met een *run-time* gedeelte dat onder andere de *constraint stores* implementeert. Uiteraard is de tijdscomplexiteit van een CHR programma sterk afhankelijk van de kwaliteit van zowel de compiler als het *run-time* gedeelte.

Dankzij het gebruik van efficiënte datastructuren voor de *constraint stores* kon bijvoorbeeld het union-find algoritme met optimale complexiteit geïmplementeerd worden in het K.U.Leuven CHR systeem [58], hetgeen onmogelijk was in bijvoorbeeld de referentie-implementatie in **SICStus**.

De algemene compilatieschema's brengen heel wat overhead met zich mee op gebied van tijd- en ruimtegebruik. In sommige veelvoorkomende gevallen is een deel van die overhead overbodig. Er zijn dan ook reeds verschillende statische analyses en optimalisaties ontwikkeld om redundante gegenereerde code te elimineren of dure algemene code te vervangen door efficiënte gespecialiseerde code. Voorbeelden hiervan worden gegeven in de volgende secties.

Eigen bijdrage. De *guard simplification* optimalisatie [68] detecteert en elimineert redundante tests die impliciet of expliciet in de *guard* van CHR regels staan. De analyse bestaat uit een logische implicatietester (geschreven in CHR) en houdt rekening met eventuele mode- en typedeclaraties. Deze optimalisatie hebben we vervolgens uitgebreid met een *occurrence subsumption* analyse [67]. Samen vormen ze de basis voor de algemenere, krachtigere *guard* en *continuation* optimalisaties [66].

Toekomstig werk. De belangrijkste doelstelling van dit project is om bij te dragen tot een verdere verbetering van de *state-of-the-art* in CHR systemen – zowel de compiler als het *run-time* gedeelte. Er is op verschillende gebieden nog veel ruimte voor verbetering. Ondanks het feit dat elk algoritme met optimale tijds- en ruimtecomplexiteit geïmplementeerd kan worden in CHR [69] is de constante factor die verborgen is in de asymptotische complexiteit voor CHR implementaties in vele gevallen twee tot drie ordes van grootte hoger dan die van implementaties in een low-level taal zoals C. Het blijft ook een open vraag of alle bestaande algoritmes op een elegante en natuurlijke manier kunnen geïmplementeerd worden in CHR.

3.2.3 Language design

Het verbeteren en uitbreiden van de *language design* van CHR maakt programmeren in CHR (nog) gemakkelijker, aangenamer en sneller. Bovendien kunnen ze (verbeteringen van) optimalisaties toelaten. Verschillende uitbreidingen aan het taal-ontwerp van CHR zijn reeds voorgesteld. In [59] werden bijvoorbeeld optionele mode-declaraties geïntroduceerd die *hash-table constraint stores* toelieten, geïndexeerd op *ground* argumenten.

Eigen bijdrage. In [65] en [67] werden eerste stappen gezet in de richting van (optioneel) getypeerde CHR constraints. De type-declaraties worden tot nu toe echter enkel gebruikt om optimalisaties te verscherpen; statische en/of dynamische type-*checking* ontbreekt nog quasi volledig[2].

Toekomstig werk. Ervaringen in andere programmeertalen geven aan dat type-checking een sterk hulpmiddel is om programmeerfouten zo vroeg mogelijk op te sporen. Bovendien kunnen voor bepaalde types, zoals bijvoorbeeld (dense) gehele getallen, specifiekere en efficiëntere *constraint store* implementaties gebruikt worden. Het implementeren van een expressief en volledig type-systeem in CHR – meer algemeen het bestuderen, verfijnen en uitbreiden van de *language design* van CHR – zal de expressiviteit en flexibiliteit van CHR verhogen, het ontwikkelen van CHR programma's vergemakkelijken, en optimalisaties toelaten die de performantie van CHR programma's verbeteren. Er zijn verschillende

[2] Een zeer beperkte vorm van statische type-checking is er al wel: de *guard* optimalisatie zal concluderen uit onjuist getypeerde argumenten van *head* constraints dat de overeenkomstige CHR regel nooit kan uitgevoerd worden; zo'n regels worden niet gecompileerd en onze compiler geeft een waarschuwing.

mogelijke verbeteringen en uitbreidingen in de *language design* van CHR mogelijk, voornamelijk op het gebied van nieuwe declaraties, compiler-directieven en -opties, syntactische suiker voor standaard-constructies en ondersteuning voor een overzichtelijke organisatie van grote CHR programma's via modules en/of constraint-hiërarchieën.

3.3 Analyse

3.3.1 Terminatie

Terminatie is in het algemeen onbeslisbaar, maar door bestaande methodes (uit gerelateerd onderzoek naar term-herschrijfsystemen en logisch programmeren) aan te passen is het mogelijk om terminatie aan te tonen voor CHR-programma's met enkel simplificatieregels [28]. Het is daarbij wel nodig om een geschikte *rangfunctie* (ook orde, norm of level-mapping genoemd) te vinden – hetgeen in sommige gevallen niet triviaal is. Soms volstaat het zelfs niet om een geschikte rangfunctie te vinden en moet bepaalde kennis over relaties tussen de argumenten van predicaten gebruikt worden.

Toekomstig werk. Terminatie voor CHR-programma's die ook propagatieregels bevatten is nog grotendeels onverkend en de gekende "level-mapping"-methode is niet eenvoudig aan te passen naar dit soort programma's. Ook het automatiseren van terminatiebewijzen voor CHR-programma's is een mogelijkheid voor verder onderzoek. Tenslotte dient de toepasbaarheid van technieken uit het domein van herschrijfsystemen onderzocht te worden.

3.3.2 Confluentie

Het resultaat van een berekening van een gegeven query heeft ongeacht de uitvoeringsstrategie steeds dezelfde logische betekenis, maar het is niet gegarandeerd dat het resultaat ook syntactisch steeds hetzelfde is. *Confluentie* [1, 6, 5] van een programma garandeert dat elke berekening die begint in dezelfde initiële toestand resulteert in dezelfde eindtoestand – m.a.w. de volgorde van regel-toepassingen en van de constraints in de query heeft geen invloed.

Toekomstig werk. Er bestaat een beslisbare, nodige en voldoende confluentietest voor eindigende CHR-programma's. Deze bestaande confluentie-analyse is echter praktisch weinig bruikbaar. Bepaalde (externe) voorwaarden, zoals beperkingen op vorm van de query of kennis over de uitvoeringsvolgorde, betekenen dat veel strikt genomen niet-confluente programma's toch niet problematisch zijn. In de praktijk [58] blijkt dat vaak het grootste deel van de confluentieproblemen die door de confluentietest gevonden worden, verworpen mogen worden wegens niet relevant gegeven een correct gebruik van het programma. Een mechanisme om de gebruiker toe te laten bijkomende eigenschappen te specifiëren zodat o.a. de confluentietest aan de hand daarvan minder overbodige "problemen" detecteert is essentieel. Het is bijvoorbeeld wenselijk dat er declaraties voorzien worden voor de algemene vorm van queries, zodat de confluentie-analyse en andere analyses en optimalisaties hiermee kunnen rekening houden. Bovendien kunnen bestaande confluentie-analyses voor de verfijnde semantiek enkel programma's met *ground* constraints behandelen, terwijl veel programma's variabelen gebruiken. Het is belangrijk dat deze *groundness*-voorwaarde gerelaxeerd kan worden om confluentie van een grotere klasse van programma's te kunnen bepalen.

3.3.3 Vervollediging (completion)

Vervollediging is het toevoegen van regels aan een niet-confluent CHR-programma tot het confluent wordt. In [4] wordt een dergelijke methode geïntroduceerd. Vervollediging kan de inconsistentie van een CHR-programma blootleggen en kan ook gebruikt worden om generische antwoorden te berekenen, nieuwe constraints te definiëren uit bestaande en daar constraint solvers voor af te leiden. Dit laatste kan de CHR-programmeur helpen om bestaande solvers uit te breiden, te wijzigen of te specialiseren.

Toekomstig werk. Een interessant onderwerp om te bestuderen is de relatie tussen vervollediging en gedeeltelijke evaluatie (partial evaluation). Automatische vervollediging kan, net zoals een automatische confluentie-checker, een handige *tool* zijn in een toekomstige programmeer-omgeving voor CHR.

3.3.4 Operationele equivalentie

Er is een beslisbare, nodige en voldoende syntactische voorwaarde voor operationele equivalentie van eindigende, confluente CHR-programma's [7].

Toekomstig werk. Het is erg belangrijk dat de klasse van CHR-programma's wordt uitgebreid waarvoor een nodige en voldoende syntactische voorwaarde voor operationele equivalentie kan gegeven worden. Ook het verband tussen operationele en logische equivalentie van CHR-programma's dient verder onderzocht te worden. Een automatische equivalentie-checker zou ook een onderdeel kunnen vormen van een toekomstige programmeer-omgeving.

3.3.5 Complexiteit

In [29, 30] wordt een algemeen resultaat voor de tijdscomplexiteit van bepaalde CHR-programma's in een (naïeve) implementatie van CHR gegeven. Hieruit blijkt dat de complexiteit gedomineerd wordt door de kost van het uitproberen van regels.

Eigen bijdrage. We bestudeerden in [69] de berekenbaarheid en complexiteit van een theoretische CHR-machine, en we vergeleken die met de Turing machine en de Random Access Memory (RAM) machine. We toonden aan dat CHR een Turing-complete taal is en dat elk RAM programma geschreven kan worden als een CHR programma dat door huidige *state-of-the-art* CHR systemen met dezelfde tijds- en ruimtecomplexiteit uitgevoerd kan worden op een RAM machine als het oorspronkelijk RAM programma. Alle algoritmes kunnen dus met optimale complexiteit in CHR geïmplementeerd worden.

Toekomstig werk. De bovengrens voor de tijdscomplexiteit zoals gegeven wordt in [29] is in veel gevallen een nogal ruwe benadering die scherper gemaakt kan worden door rekening te houden met de operationele semantiek van implementaties die regels toepassen in de volgorde zoals ze in het programma voorkomen. Indexeren van bepaalde constraints kan de kost van het uitproberen van regels drastisch verminderen. Declaraties van de gebruiker en/of automatische analyses van eigenschappen van constraints kunnen in veel gevallen de tijd- en ruimtecomplexiteit verder reduceren.

Een algemeen theoretisch kader om de complexiteit te analyseren van CHR-programma's die in een geavanceerde, optimaliserende implementatie uitgevoerd worden ontbreekt momenteel echter. Propagatieregels – een vorm van *fixpoint*-berekeningen – zijn een bijkomend probleem voor complexiteitsanalyse. In de loop van dit project hopen we nieuwe inzichten te verwerven over deze problematiek en op basis daarvan nieuwe complexiteitsanalyses te ontwikkelen.

3.4 Optimalisaties

Bestaand werk. We sommen enkele optimalisaties op, waarvan sommige reeds geïmplementeerd zijn in bestaande CHR systemen. In [42, 23] worden enkele van deze optimalisaties beschreven.

- **Join ordering** is het herschikken van de constraints in de *head* van een regel aan de hand van een heuristiek zodat het zoeken naar *partner constraints* om de toepasbaarheid van de regel te testen efficiënter kan gebeuren. Stel bijvoorbeeld dat `p` de actieve constraint is en de toepasbaarheid van de regel `p(X),q(Y,Z),r(X,Y) <=> G | B` getest moet worden. In deze volgorde worden eerst *alle* `q/2`-constraints uitgeprobeerd. Als we de regel eerst hadden herschreven naar de equivalente regel `p(X),r(X,Y),q(Y,Z) <=> G | B`, dan zou dankzij `r/2` de variabele `Y` eerst gebonden worden en moeten veel minder `q/2`-constraints uitgeprobeerd worden.

- **Functionele afhankelijkheden** tussen argumenten van constraints kunnen (indien gekend) het zoeken naar constraints versnellen bij het testen of een regel toepasbaar is.

- **Verzameling-semantiek.** Hoewel CHR in het algemeen een multi-set semantiek heeft (dezelfde constraint kan verschillende keren voorkomen) is voor veel constraints gedefinieerd in CHR-programma's het aantal kopieën onbelangrijk. Als er bijvoorbeeld een regel voorkomt als `p \ p <=> true` is dit het geval. Constraints met deze semantiek kunnen gedetecteerd worden en efficiënter geïndexeerd worden.

- **Symmetrie** komt regelmatig voor in CHR-programma's. Een constraint die gelijkheid of ongelijkheid van zijn argumenten uitdrukt is bijvoorbeeld symmetrisch. De gecompileerde code kan compacter en efficiënter gemaakt worden voor constraints waarvan de symmetrie kan gedetecteerd worden.

- **Efficiëntere datastructuren** zoals hash-tables kunnen gebruikt worden als bepaalde eigenschappen van constraints gekend zijn, zoals groundness (cfr. sectie 3.4.1).

- **Delay avoidance** aan de hand van antimonotoniciteit [54] reduceert het aantal variabelen die een constraint kunnen triggeren – hetgeen op zich al een zekere overhead elimineert – zodat zo weinig mogelijk overbodige (niet resulterend in bijkomende regeltoepassingen) triggerings van constraints gebeuren.

- **Programma-specialisatie** is een programma-transformatie waardoor CHR-regels partieel geëvalueerd worden met betrekking tot een gegeven query. De regels die op die manier toegevoegd worden, kunnen de performantie verbeteren van een CHR-programma.

- **Never-stored analyse** probeert te detecteren welke constraints nooit zullen worden opgeslagen in de constraint store. Als aan het voorbeeld uit sectie 3.1.1 bijvoorbeeld de regel `geq(X,Y) <=> leq (Y,X)` wordt toegevoegd, dan is het duidelijk dat `geq/2`-constraints altijd onmiddellijk gesimplificeerd kunnen worden. Never-stored constraints kunnen naar efficiëntere code gecompileerd worden.

Eigen bijdrage. Zoals reeds vermeld in sectie 3.2.2 heb ik al enkele bijdragen geleverd in de context van optimalisatie van CHR programma's:

- **Guard optimalisatie** [66] elimineert redundante *guard*-testen. Ook impliciete testen zoals *head matching* kunnen worden geschrapt of vereenvoudigd. Door eerst deze optimalisatie door te voeren kan de never-stored analyse meer gevallen detecteren. Deze optimalisatie is geïmplementeerd in de K.U.Leuven CHR compiler.

- **Continuation optimalisatie** [66] verandert de uitvoeringsstrategie in de gecompileerde code zodat het uitproberen van regels die zeker niet kunnen toegepast worden overgeslagen wordt. Deze optimalisatie is nog niet geïmplementeerd.

Toekomstig werk. In het algemeen zijn verschillende van bovenstaande optimalisaties ofwel nog niet geïmplementeerd, ofwel is enkel een afgezwakte versie geïmplementeerd die slechts een conservatieve benadering van de optimaliseerbare gevallen kan detecteren. In dit project zullen we onderzoeken hoe voor verschillende optimalisaties de klasse van detecteerbare optimaliseerbare constructies kan uitgebreid worden.

In de volgende sectie bespreken we een framework voor abstracte interpretatie, waarbinnen we deze en andere analyses/optimalisaties kunnen formuleren en er op die manier meer inzicht in verwerven.

3.4.1 Abstracte interpretatie

Abstracte interpretatie (AI) [16, 10, 11, 9, 47] is een manier om op een formele manier conservatieve benaderingen te construeren van de semantiek van een programma. In de praktijk kan AI gebruikt worden om analyse-algoritmen te construeren die automatisch, statisch en conservatief dynamische eigenschappen van programma's kunnen afleiden. Dergelijke eigenschappen van het *run-time* gedrag van programma's zijn nuttig voor debugging, type-inferentie, code optimalisatie, programma-transformatie en correctheidsbewijzen zoals terminatie- of confluentiebewijzen.

Een abstracte interpretatie wordt gedefinieerd als een niet-standaard, benaderende semantiek die bekomen wordt uit de standaard semantiek door het concrete berekeningsdomein en de overeenkomstige concrete semantische operaties te vervangen door een abstract domein met overeenkomstige abstracte semantische operaties. Een abstractiefunctie α moet gedefinieerd worden die een concrete toestand afbeeldt op een abstracte toestand, en een concretisatiefunctie γ die een abstracte toestand afbeeldt op een verzameling van concrete toestanden.

Bestaand werk. Bestaande AI-frameworks – zoals voor **Prolog** [10, 9] – zijn in het algemeen niet (rechtstreeks) bruikbaar (of leiden tot een onaanvaardbaar verlies aan precisie) voor CHR omdat daar geen rekening gehouden wordt met de specifieke kenmerken van CHR. Recentelijk is onderzoek begonnen naar een AI-framework voor CHR. In [60] wordt een algemeen AI-framework voor CHR beschreven en gebruikt in twee analyses: groundness-analyse en storage-analyse.

- **Groundness-analyse.** De abstractie die gemaakt wordt in groundness-analyse is het al dan niet *ground* zijn van variabelen en argumenten van constraints. Er zijn tal van optimalisaties mogelijk gebaseerd op deze informatie:

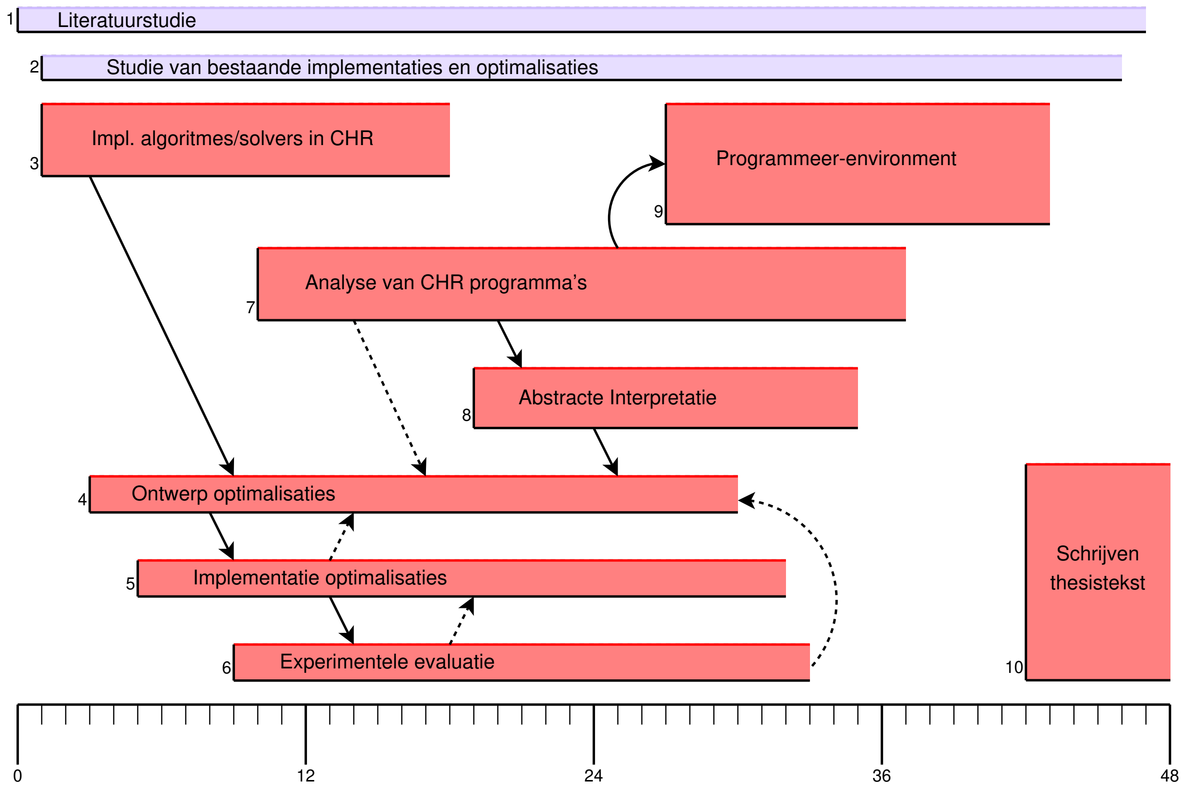
  - *Representatie-specialisatie:* Een constraint waarvan de argumenten *ground* zijn kan niet getriggered worden door een ingebouwde constraint. Dit betekent dat voorzieningen daarvoor overbodig zijn in de representatie van dergelijke constraints. Zo kunnen bijvoorbeeld efficiëntere datastructuren – zoals *hashtables* in het K.U.Leuven-CHR-systeem – gebruikt worden om die constraints in op te slaan.
  - *Guard-specialisatie:* De semantiek van CHR laat niet toe dat condities in de *guard* ervoor zorgen dat variabelen gebonden worden. Huidige CHR-implementaties controleren dynamisch dat dit niet gebeurt. Met de hulp van groundness-informatie kunnen redundante run-time controles vermeden worden.
  - *Verbetering van andere analyses:* In het algemeen kan groundness-informatie nuttig zijn om de resultaten van andere analyses – zoals bijvoorbeeld de storage-analyse, zie hieronder – te verbeteren.

- **Storage-analyse.** Volgens de standaard (verfijnde) operationele semantiek moet een constraint onmiddellijk toegevoegd worden als hij bovenaan de uitvoerings-stack staat. We willen dit vermijden indien dat geen verschil maakt voor de uiteindelijke eindtoestand. In het algemeen is het namelijk beter om een constraint zo laat mogelijk in de *constraint store* op te slaan. Als een constraint namelijk verwijderd wordt voor hij opgeslagen wordt, dan kan de overhead van het toevoegen aan en verwijderen uit de *constraint store* vermeden worden. Aan de hand van de resultaten van *storage*-analyse kan zoveel mogelijk dergelijke overhead vermeden worden.

Toekomstig werk. In dit project zullen we enerzijds nieuwe analyses binnen het AI-framework ontwikkelen en anderzijds een beter inzicht trachten te verwerven in bestaande analyses door die te formuleren in het AI-framework, om op die manier bestaande analyses te vervangen of te verbeteren. We hopen ook een beter inzicht te verkrijgen in de trade-offs die gemaakt worden, bijvoorbeeld tussen de nauwkeurigheid van de analyse en de efficiëntie van de uitvoering ervan.

4 Planning

Het voorgestelde onderzoeksproject bestaat uit een aantal deeltaken, waarvan de planning aangegeven is in onderstaande figuur. De horizontale as stelt het tijdsverloop in maanden voor. Pijlen tussen verschillende deeltaken stellen afhankelijkheden voor. Om de figuur niet onnodig ingewikkeld te maken zijn de triviale pijlen van 1 en 2 naar alle andere deeltaken en van alle deeltaken naar 10 weggelaten.



Figuur 1: Planning van het onderzoeksproject

1. **Literatuurstudie** (3 maanden gespreid over de gehele duur van het project). Diepgaande studie van de wetenschappelijke literatuur in het CHR-domein, maar ook in aanverwante domeinen zoals *term-rewrite systems* en CLP. Gedurende het gehele onderzoeksproject blijft literatuurstudie belangrijk.

2. **Studie van bestaande implementaties en optimalisaties** (3 maanden gespreid over 45 maanden). Gedetailleerde studie van de bestaande CHR-implementaties en analyse- en optimalisatietechnieken die daarin gebruikt worden. Ook hier dienen de evoluties gevolgd te worden gedurende het volledige project.

3. **Implementeren van algoritmes en solvers in CHR** (5 maanden gespreid over 17 maanden). Identificeren van mogelijke *language design* verbeteringen en *performance bottlenecks*.

4. **Ontwerpen van optimalisaties (en nieuwe taal-elementen)** (5 maanden gespreid over 27 maanden). Bruikbaarheid nagaan en ontwerpen van verschillende

nieuwe optimalisaties en nieuwe taal-elementen (extra declaraties, syntactische suiker, nieuwe built-ins, ...), gebaseerd op de ervaringen uit het vorige punt. Zie secties 3.4 en 3.2.3.

5. **Implementeren van optimalisaties** (4 maanden gespreid over 27 maanden). Implementatie van bovenstaande optimalisaties en taal-uitbreidingen. Zie secties 3.2 en 3.4.

6. **Experimentele evaluatie van optimalisaties** (4 maanden gespreid over 24 maanden). Verificatie en evaluatie van deze optimalisaties: enerzijds correctheid (indien mogelijk formeel) aantonen, anderzijds performantiewinst meten d.m.v. *benchmarks* en experimenten met programma's op *real-life*-schaal.

7. **Analyse van CHR programma's** (8 maanden gespreid over 27 maanden). Theoretische studie van allerlei eigenschappen van CHR programma's. Zie sectie 3.3.

8. **Abstracte Interpretatie** (4 maanden gespreid over 16 maanden). Verder ontwikkelen van een algemeen theoretisch kader voor abstracte interpretatie van CHR-programma's, waarbinnen geredeneerd kan worden over en correctheid kan bewezen worden van de analyses en optimalisaties uit de vorige punten. Zie sectie 3.4.1.

9. **Programmeer-environment** (6 maanden gespreid over 16 maanden). Ontwikkelen van allerlei *tools* die programmeurs kunnen helpen bij het schrijven van CHR programma's. Verschillende potentiële *tools* zijn automatische *checkers* van de analyses ontwikkeld in punt 7.

10. **Schrijven van de doctoraatsthesis** (6 maanden).

5 Toepassingsmogelijkheden

CHR is een programmeertaal(-uitbreiding) die in een hele waaier van toepassingen zeer nuttig gebleken is. Vooral voor toepassingen waar eenvoudige aanpassing aan veranderende probleemomschrijvingen cruciaal is – zoals de *Munich Rent Advisor* waarvan de probleemspecificatie bij elke nieuwe editie van de *Mietspiegel* wijzigt en in elke stad verschillend is – blijkt het declaratief karakter en de enorme flexibiliteit van CHR een gigantisch voordeel. Ook zowat alle soorten van *planning*- en *scheduling*-problemen zijn het best op te lossen met CLP, en aangezien ook hier de concrete probleemspecificaties typisch geregeld veranderen is de flexibiliteit van CHR echt een *must*. Enkele bestaande toepassingen van CHR:

- *The Munich Rent Advisor (MRA)* [33] is een electronische versie van de *Mietspiegel* van München, een publicatie van het stadsbestuur die in feite een expertsysteem beschrijft om de normale huurprijs van een appartement te schatten op basis van een vragenlijst. Deze schatting is juridisch bindend. Het manueel berekenen van zo'n geschatte huurprijs kan voor een leek een weekend in beslag nemen en zelfs een menselijke expert doet er ongeveer een half uur over. Dankzij de MRA is de berekeningstijd gereduceerd tot de tijd nodig om de on-line vragenlijst (gedeeltelijk) in te vullen. Deze toepassing is ontwikkeld in 1996 en geschreven in één van de eerste CHR-implementaties in **ECL^{i}PS^e Prolog**.

- *Het opstellen van lessenroosters* is een typisch *scheduling*-probleem dat met Constraint Logic Programming kan opgelost worden. In [8] wordt beschreven hoe het lessenrooster van de opleiding computerwetenschappen aan de universiteit van München automatisch werd opgesteld door een CHR-programma.

- *POPULAR – Planning Cordless Communication* [36] is een toepassing die in samenwerking met Siemens ontwikkeld werd om de optimale plaatsing van zenders voor draadloze draagbare apparaten zoals telefoons. POPULAR berekent het minimaal aantal zenders en hun plaatsing gegeven de blauwdruk van een gebouw en informatie over de materialen gebruikt in de muren en plafonds. Deze toepassing werd beschouwd als één van de meest innovatieve telecommunicatie-applicaties door IEEE Expert Magazine en won de *Telecom application contest* van Telecom Italia op de jaarlijkse conferentie over Constraint Programming (CP'98).

- *Cuypers* [39] is een systeem om automatisch web-gebaseerde presentaties te genereren als een interface naar een semi-gestructureerde multimedia-database. Het is ontworpen voor het genereren van presentaties die interactief, tijd-gebaseerd en *media-centric* zijn.

- JmmSolve [55] is een referentie-implementatie voor Concurrent Constraint-based Memory (CCM) machines [52], een framework voor **Java** geheugenmodellen. JmmSolve werd ontwikkeld door onze onderzoeksgroep.

Andere domeinen waarin CHR nuttig is gebleken zijn onder andere multi-agent systemen [64], computationele linguistiek [51] en cognitieve robotica [71].

Referenties

[1] Slim Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Proc. 3rd Intl. Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 252–266, Schloss Hagenberg, Austria, 1997.

[2] Slim Abdennadher. A Language for Experimenting with Declarative Paradigms. In *Proc. 2nd Workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, 2000.

[3] Slim Abdennadher et al. JACK: A Java Constraint Kit. *Electronic Notes in Theoretical Computer Science*, 64, 2002.

[4] Slim Abdennadher and Thom Frühwirth. On Completion of Constraint Handling Rules. In *Proc. 4th Intl. Conference on Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, 1998.

[5] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On Confluence of Constraint Handling Rules. In *Proc. 2nd Intl. Conference on Principles and Practice of Constraint Programming (CP'96)*, LNCS 1118. Springer, 1996.

[6] Slim Abdennadher, Thom Fruhwirth, and Holger Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints Journal, Special Issue on 2nd Intl. Conference on Principles and Practice of Constraint Programming*, 4(2), 1999.

[7] Slim Abdennadher and Thom Früwirth. Operational Equivalence of CHR Programs and Constraints. In *Proc. 5th Intl. Conference on Principles and Practice of Constraint Programming (CP'99)*. Springer LNCS, Alexandra, Virginia, 1999.

[8] Slim Abdennadher and Michael Marte. University course timetabling using Constraint Handling Rules. *J. Applied Artificial Intelligence*, 14(4):311–326, 2000.

[9] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *J. Logic Programming*, 10(2):91–124, 1991.

[10] Maurice Bruynooghe, Gerda Janssens, Alain Callebaut, and Bart Demoen. Abstract interpretation: Towards the global optimisation of Prolog programs. In *Proc. 1987 Symposium on Logic Programming*, pages 192–204, 1987.

[11] Maurice Bruynooghe and Danny De Schreye. Tutorial on abstract interpretation of logic programs. In *Joint Intl. Conference and Symposium on Logic Programming, JICSLP'88*, Seattle, Washington, USA, 1988.

[12] Michael Codish and Bart Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proc. Static Analysis*, LNCS 864, pages 281–296, Aachen, Germany, 1994.

[13] Michael Codish and Bart Demoen. Analysing logic programs using 'prop'-ositional logic programs and a magic wand. *J. Logic Programming*, 25(3):249–274, 1995.

[14] Michael Codish, Anne Mulkers, Maurice Bruynooghe, María García de la Banda, and Manuel Hermenegildo. Improving abstract interpretations by combining domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, 1995.

[15] Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). *J. Logic Programming*, 27(3):185–226, 1996.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[17] Luís Damas, Vítor Santos Costa, Rogério Reis, and Rúben Azevedo. *YAP User's Manual*. Universidade do Porto.

[18] María García de la Banda, Manuel Hermenegildo, Maurice Bruynooghe, Veroniek Dumortier, Gerda Janssens, and Wim Simoens. Global analysis of constraint logic programs. *ACM Trans. Programming Languages and Systems*, 18(5):564–614, 1996.

[19] Bart Demoen. The hProlog home page. *http://www.cs.kuleuven.ac.be/˜bmd/hProlog/*.

[20] Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An overview of HAL. In *Proc. Principles and Practice of Constraint Programming*, pages 174–188, 1999.

[21] Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. Herbrand constraint solving in HAL. In *Lecture Notes in Computer Science 3049*, pages 501–540, June 2004.

[22] Mehmet Dincbas et al. The constraint logic programming language CHIP. In *Proc. Intl. Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1988.

[23] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending Arbitrary Solvers with Constraint Handling Rules. In *Proc. 5th Intl. Conference on Principles and Practice of Declarative Programming*, 2003.

[24] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *20th Intl. Conference on Logic Programming (ICLP'04)*, Saint-Malo, France, September 2004.

[25] Thom Frühwirth. Introducing simplification rules. Technical Report ECRC-LP-63, ECRC, Munich, Germany, 1991.

[26] Thom Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science, pages 90–107. Springer Verlag, 1995.

[27] Thom Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *J. Logic Programming, Special Issue on Constraint Logic Programming*, volume 37(1–3), 1998.

[28] Thom Frühwirth. Proving Termination of Constraint Solver Programs. In *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus*, volume 1865 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 2000.

[29] Thom Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *KR2002: Principles of Knowledge Representation and Reasoning*, pages 547–557. San Francisco, California, 2002.

[30] Thom Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In *Quantitative Aspects of Programming Languages, Selected Papers*, volume 59(3) of *Electronic Notes in Theoretical Computer Science*. 2002.

[31] Thom Frühwirth. Specialization of Concurrent Guarded Multi-Set Transformation Rules. In *Intl. Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2004)*, Verona, Italy, 2004.

[32] Thom Frühwirth. Logical rules for a lexicographic order constraint solver. In *Proc. 2nd Workshop on Constraint Handling Rules*, Sitges, Spain, October 2005. To appear.

[33] Thom Frühwirth and Slim Abdennadher. The Munich Rent Advisor: A Success for Logic Programming on the Internet. In M. Hermenegildo, L. Naish, and L. Sterling, editors, *Theory and Practice of Logic Programming, Special Issue on Logic Programming and the Internet*, volume 1(3), 2001.

[34] Thom Frühwirth and Pascal Brisset. *Chapter on Constraint Handling Rules.* released by ECRC, Munich, Germany, 1995.

[35] Thom Frühwirth and Pascal Brisset. High-level implementations of Constraint Handling Rules. Technical Report ECRC-95-20, ECRC, 1995.

[36] Thom Frühwirth and Pascal Brisset. Optimal Placement of Base Stations in Wireless Indoor Telecommunication. In *Proc. 4th Intl. Conference on Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, 1998.

[37] Thom Frühwirth et al. Constraint logic programming - an informal introduction. In *Logic Programming in Action.* Springer LNCS 636, 1992.

[38] Thom Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic Constraint Handling Rules. In *11th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, volume 76 of *Electronic Notes in Theoretical Computer Science*, 2002.

[39] Joost Geurts, Jacco van Ossenbruggen, and Lynda Hardman. Application-specific constraints for multimedia presentation generation. In *Proc. Intl. Conference on Multimedia Modeling 2001*, pages 247–266, Amsterdam, The Netherlands, 2001.

[40] Pascal Van Hentenryck. Constraint Logic Programming. *The Knowledge Engineering Review*, 6(3):151–194, 1991.

[41] Bernhard Herbig. Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus. Master's thesis, University of Kaiserslautern, Germany, 1993.

[42] Christian Holzbaur, María García de la Banda, David Jeffery, and Peter J. Stuckey. Optimizing Compilation of Constraint Handling Rules. In *Proc. 17th Intl. Conference on Logic Programming (ICLP)*, pages 74–89. LNCS, Springer-Verlag, 2001.

[43] Christian Holzbaur and Thom Frühwirth. CHR reference manual. Technical Report TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, 1998.

[44] Christian Holzbaur and Thom Frühwirth, editors. *Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, volume 14(4). Taylor & Francis, 2000.

[45] Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *J. Logic Programming*, 19/20:503–581, 1994.

[46] Gerda Janssens. *Deriving run time properties of logic programs by means of abstract interpretation.* PhD thesis, Dept. Computer Science, K.U.Leuven, Belgium, 1990.

[47] Gerda Janssens and Maurice Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Programming*, 13(2&3):205–258, 1992.

[48] Gerda Janssens and Wim Simoens. On the implementation of abstract interpretation systems for constraint logic programs. In *Proceedings of the 5th International Conference on Compiler Construction*, pages 172–187, 1994.

[49] Antonis C. Kakas, Bert Van Nuffelen, and Marc Denecker. A-system : Problem solving through abduction. In *Proc. 17th Intl. Joint Conference on Artificial Intelligence*, pages 591–596. Morgan Kaufmann Publishers, Inc, 2001.

[50] Bert Van Nuffelen and Antonis C. Kakas. A-System : Programming with abduction. In *Proc. Logic Programming and Nonmonotonic Reasoning*, pages 393–396, 2001.

[51] Gerald Penn. Applying Constraint Handling Rules to HPSG. In *Proc. 1st Intl. Conference on Computational Logic, Workshop on Rule-Based Constraint Reasoning and Programming*, UK, 2000. Imperial College.

[52] Vijay Saraswat. Concurrent Constraint-based Memory Machines: A framework for Java Memory Models. Technical report, IBM TJ Watson Research Center, 2004.

[53] Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules.* PhD thesis, Dept. Computer Science, K.U.Leuven, Belgium, 2005.

[54] Tom Schrijvers and Bart Demoen. Antimonotony-based Delay Avoidance for CHR. Technical Report CW 385, K.U.Leuven, Dept. Computer Science, July 2004.

[55] Tom Schrijvers and Bart Demoen. JmmSolve: a generative reference implementation of CCM machines. Technical Report CW 379, K.U.Leuven, Dept. Comp. Sc., 2004.

[56] Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *Proc. 1st Workshop on Constraint Handling Rules: Selected Contributions*, Ulm, Germany, May 2004. Website at *http://www.cs.kuleuven.ac.be/˜toms/Research/CHR/.*

[57] Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic Implication Checking for CHR Constraints. In *Proc. 6th Intl. Workshop on Rule-Based Programming*, Nara, Japan, April 2005.

[58] Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 2005. To appear.

[59] Tom Schrijvers and Thom Früwirth. Implementing and Analysing Union-Find in CHR. Technical Report CW 389, K.U.Leuven, Dept. CS, July 2004.

[60] Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *PPDP'05: Proc. 7th Intl. Symposium on Principles and Practice of Declarative Programming*, Lisbon, Portugal, July 2005. ACM Press.

[61] Tom Schrijvers and David S. Warren. Constraint Handling Rules and tabled execution. In Bart Demoen and Vladimir Lifschitz, editors, *Proc. 20th Intl. Conference on Logic Programming*, pages 120–136, St-Malo, France, September 2004.

[62] Tom Schrijvers, David S. Warren, and Bart Demoen. CHR for XSB. In Ricardo Lopes and Michel Ferreira, editors, *Proc. CICLOPS 2003. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto*, pages 7–20, December 2003.

[63] Tom Schrijvers, Jan Wielemaker, and Bart Demoen. Poster: Constraint Handling Rules for SWI-Prolog. In Armin Wolf, editor, *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, Ulm, Germany, February 2005.

[64] Christian Seitz, Bernhard Bauer, and Michael Berger. Planning and Scheduling in Multi Agent Systems using Constraint Handling Rules. In *Intl. Conference on Artificial Intelligence (IC-AI' 02)*, Las Vegas, Nevada, USA, 2002. CSREA Press.

[65] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. Technical Report CW 396, K.U.Leuven, Dept. CS, November 2004.

[66] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard and Continuation Optimization for Occurrence Representations of CHR. In *Proc. 21st Intl. Conference on Logic Programming (ICLP'05)*, Sitges, Spain, October 2005. To appear.

[67] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Reasoning for CHR Optimization. Technical Report CW 411, K.U.Leuven, Dept. CS, May 2005.

[68] Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Simplification in CHR programs. In *Proc. 19th Workshop on (Constraint) Logic Programming*, number 2005-01 in Ulmer Informatik-Berichte, pages 123–134, Ulm, Germany, February 2005.

[69] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *Proc. 2nd Workshop on Constraint Handling Rules*, Sitges, Spain, October 2005. To appear.

[70] Martin Sulzmann. TIE: A CHR-based Type Inference Engine. Technical Report TR2001/27, Dept. Computer Science, The University of Melbourne, 2001.

[71] Michael Thielscher. Reasoning about actions with CHRs and finite domain constraints. In P. Stuckey, editor, *Proc. Intl. Conference on Logic Programming (ICLP)*, volume 2401 of *LNCS*, pages 70–84, Copenhagen, Denmark, 2002.

[72] David S. Warren et al. The XSB home page. *http://xsb.sourceforge.net*.

[73] Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a User-Friendly, Flexible and Efficient CHR System for Java. In *Proc. 2nd Workshop on Constraint Handling Rules*, Sitges, Spain, October 2005. To appear.

[74] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebrenik, editors, *Proc. 13th Intl. Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, 2003.

[75] Armin Wolf. Adaptive Constraint Handling with CHR in Java. In *Proc. 7th Intl. Conference on Principles and Practice of Constraint Programming*, 2001.