Aanvraag specialisatiebeurs IWT

Beschrijving van het onderzoeksproject

"Analyse en Optimalisatie van CHR-programma's"

Jon Sneyers

Inhoudsopgave

1 Probleemstelling **1**

2 Doelstelling **2**

3 Projectbeschrijving **3**
3.1 Inleiding: Constraint Handling Rules . . . 3
3.1.1 Voorbeeld . . . 3
3.1.2 Syntax . . . 3
3.1.3 Declaratieve semantiek . . . 4
3.1.4 Operationele semantiek . . . 5
3.2 Analyse . . . 5
3.2.1 Correctheid en volledigheid . . . 6
3.2.2 Terminatie . . . 6
3.2.3 Confluentie . . . 7
3.2.4 Vervollediging (completion) . . . 8
3.2.5 Operationele equivalentie . . . 8
3.2.6 Complexiteitsanalyse . . . 8
3.3 Optimalisaties . . . 9
3.3.1 Bestaand werk . . . 9
3.3.2 Eigen bijdrage . . . 10
3.4 Abstracte interpretatie . . . 11
3.4.1 Bestaand werk . . . 11
3.4.2 Eigen bijdrage . . . 12
3.5 Implementatie van CHR . . . 12
3.5.1 Bestaand werk . . . 12
3.5.2 Eigen bijdrage . . . 12

4 Planning **13**

5 Toepassingsmogelijkheden **15**

Referenties **16**

1 Probleemstelling

Constraint Logic Programming [45, 59, 60, 61] (CLP) combineert de voordelen van logisch programmeren en *constraint solving*. 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 solving* worden efficiënte probleem-specifieke algoritmes gebruikt om deelproblemen op te lossen waarin relaties voorkomen die beperkingen (*constraints*) uitdrukken. Een *constraint solver* kan gezien worden als een inferentie-systeem dat basisoperaties op constraints ondersteunt.

In de begindagen van CLP werd gebruik gemaakt van de zogenaamde "black-box"-aanpak, waarbij een ingebouwde constraint solver geïmplementeerd werd in een low-level programmeertaal. Deze aanpak resulteert in een efficiënte solver maar maakt het erg moeilijk om een solver te bouwen over een nieuw domein of een bestaande solver aan te passen, laat staan te debuggen of te analyseren. Verscheidene "glass-box"-technieken zijn sindsdien voorgesteld [29, 48, 49, 12, 79, 20, 51] om constraint systemen beter aanpasbaar te maken, maar die technieken laten typisch enkel toe om een solver over een gegeven (meestal eindig) constraint domein uit te breiden. Rechtstreeks implementeren van solvers over een domein dat specifiek is voor de toepassing is vaak onmogelijk of onpraktisch.

Constraint Handling Rules [35, 36] (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 zo 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 verschillende programmeertalen, onder andere in Java [9, 11, 86] en in een half dozijn Prolog-varianten [34, 43, 42, 54, 22, 75] (zie sectie 3.5.1 voor een bondig overzicht). Over de jaren werd duidelijk dat CHR en uitbreidingen ervan [2] nuttig zijn voor de implementatie van redeneersystemen in het algemeen [57].

Het belangrijkste nadeel van huidige CHR-implementaties ten opzichte van low-level implementaties van specifieke constraint solvers is de lagere efficiëntie van CHR, onder andere door een gebrek aan gebruikercontrole over de manier waarop CHR-programma's gecompileerd worden. CHR-programma's, die door een algemene CHR-compiler gecompileerd en in een algemene run-time omgeving uitgevoerd worden, zijn hierdoor vaak enkel bruikbaar voor proto-typing, waarbij een gespecialiseerde en in een low(er)-level programmeertaal geïmplementeerde solver achteraf nog steeds moet geschreven worden om in de praktijk bruikbaar te zijn. Het zou wenselijk zijn dat CHR-programma's qua efficiëntie in de buurt komen van low-level-geïmplementeerde solvers. Om dit te bereiken zullen echter verregaande optimalisaties moeten worden ontwikkeld en geïmplementeerd.

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* [36, 1, 7, 32] en *equivalentie* [8], *terminatie* [37], *confluentie* [1, 7, 6, 3] en *complexiteit* [38, 39]. Verder onderzoek naar deze en andere eigenschappen is wellicht nodig. Bestaande optimalisatietechnieken voor CHR-programma's [52, 4, 73, 40, 31] dienen te worden verfijnd en nieuwe optimalisaties moeten worden ontwikkeld en geïmplementeerd.

2 Doelstelling

De belangrijkste doelstelling van dit project is het ontwerpen, implementeren en publiceren van nieuwe analyses en optimalisaties voor CHR-programma's, alsmede het aanbrengen van verbeteringen aan bestaande analyses en optimalisaties. We hopen diepere inzichten te verwerven in bestaande analyses door ze te bestuderen binnen een algemeen framework voor abstracte interpretatie van CHR-programma's. Eerste stappen zijn reeds gezet in de ontwikkeling van een dergelijk framework [30] en in dit project zal de ontwikkeling hiervan voortgezet worden.

Een andere doelstelling is het bestuderen en verbeteren van bestaande CHR-implementaties en het integreren van analyses in de *bootstrapped* CHR-compiler die momenteel in ontwikkeling is en die de *portability* van CHR naar verschillende platformen drastisch zal verhogen. Onze aandacht zal voornamelijk gaan naar het Java-platform en logische programmeertalen. In de marge van het onderzoek zal ook enige aandacht gegeven worden aan de *language design* van CHR, meer bepaald kunnen we nieuwe syntactische elementen voorstellen zoals type- en andere declaraties. Ook een gebruiksvriendelijke *environment* kan worden ontwikkeld, 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 gemakkelijker.

In het algemeen is het onze doelstelling om de bruikbaarheid van CHR te verhogen, enerzijds op het gebied van efficiëntie, anderzijds op het gebied van gebruiksvriendelijkheid van het schrijven van correcte CHR-programma's. De analyses die in de volgende sectie besproken worden, zijn nuttig als basis voor optimalisaties (efficiëntieverhoging), maar ook als programmeer-*tools* in een gebruiksvriendelijke *environment*.

Onze onderzoekseenheid heeft in het verleden al veel belangrijke bijdragen geleverd in de domeinen van analyse [15, 23, 71, 63, 28, 16, 19, 18, 17], frameworks voor analyse [62, 14, 63, 65, 64], en toepassingen van Constraint Logic Programming [68, 66, 65, 27].

Ook specifiek binnen het recente domein van Constraint Handling Rules is er binnen onze onderzoekseenheid een behoorlijke expertise [76, 73, 77, 72, 30, 74, 83]. Het K.U.Leuven CHR-systeem [75] is een state-of-the-art implementatie van CHR die momenteel voor drie Prolog-omgevingen beschikbaar is: hProlog [25], SWI-Prolog [85] en XSB [84]. Dit systeem werd ontwikkeld door Tom Schrijvers en Bart Demoen, die beide 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) in de afdeling Informatica van het departement Computerwetenschappen aan de faculteit Toegepaste Wetenschappen van de K.U.Leuven. Onze onderzoeksgroep onderhoudt tevens contacten met buitenlandse onderzoeksgroepen zoals

- de onderzoeksgroepen van prof. Peter J. Stuckey en van prof. Kim Marriott [30, 26, 27], beide in Australië, respectievelijk aan de universiteit van Melbourne en aan Monash University
- de onderzoeksgroep van prof. David S. Warren [76, 77] aan de universiteit van Stony Brook, New York, USA
- de onderzoeksgroep van prof. Thom Frühwirth [83], 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. Vervolgens, in sectie 3.2, geven we een overzicht van bestaande analyses, die onder andere nuttig zijn voor allerlei optimalisaties, besproken in sectie 3.3. In sectie 3.4 wordt abstracte interpretatie besproken. Tenslotte bespreken we in sectie 3.5 de implementatie van CHR in *host*-talen.

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 [36].

3.1.1 Voorbeeld

Laat ons een gebruiker-gedefinieerde constraint implementeren 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. 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 CHR*), 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 CHR*).

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`. Aangezien `B=C` kan `antisymmetrie` nogmaals toegepast worden, dit keer op `A=<B,C=<A`. Nu is de query opgelost (met antwoord `A=B,B=C`) want er zijn enkel ingebouwde constraints over.

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 gebruiker-gedefinieerde 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 ingebouwde constraints.

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:*

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

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

[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.

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

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 operationele semantiek van CHR-programma's wordt gegeven door een transitiesysteem 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* matcht 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* matcht 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 $S \mapsto^\star S'$

Opmerking: Bijna alle huidige implementaties van CHR volgen een striktere, *verfijnde* semantiek [32] 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.

3.2 Analyse

In deze sectie presenteren we enkele resultaten die van belang zijn voor analyse van CHR-programma's. Aantonen van de correctheid van optimalisaties (cfr. sectie 3.3) gebeurt typisch door te laten zien dat eigenschappen als terminatie, confluentie en operationele equivalentie bewaard blijven. We geven ook mogelijkheden aan voor verder onderzoek.

3.2.1 Correctheid en volledigheid

Definitie 3.3 (Berekening) *Een* berekening van een goal G *is een sequentie* $S_0, S_1, \ldots$ *van toestanden met* $S_i \mapsto S_{i+1}$ *en waarbij* $S_0 = \langle G, \texttt{true}, \texttt{true} \rangle$ *de initiële toestand is. Een berekening eindigt in een eindtoestand of divergeert.*

Stelling 3.1 (Correctheid) *Laat* P *een CHR-programma zijn en* G *een query. Als* G *een berekening heeft met antwoord* C *dan*

$$\mathcal{P}, CT \models \forall(C \leftrightarrow G)$$

Stelling 3.2 (Volledigheid) *Laat* P *een CHR-programma zijn,* G *een query met minstens één eindige berekening en* C *een conjunctie van constraints. Als* $\mathcal{P}, CT \models \forall(C \leftrightarrow G)$*, dan heeft* G *een berekening met antwoord* C' *zodat*

$$\mathcal{P}, CT \models \forall(C \leftrightarrow C')$$

Stelling 3.3 (Volledigheid van gefaalde berekeningen) *Laat* P *een correct (*$\mathcal{P} \cup CT$ *consistent) CHR-programma zijn,* G *een* data-sufficiente *query (heeft een berekening met eindtoestand van de vorm* $\langle \texttt{true}, E, D \rangle$*). Als* $\mathcal{P}, CT \models \neg\exists G$*, dan heeft* G *een eindig falende berekening.*

3.2.2 Terminatie

Bestaand werk. 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 [37]. 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.

Definitie 3.4 (Rangfunctie) *Laat* f *een functie- of predicaatsymbool met ariteit* n *zijn en* t_i *(*$1 \leq i \leq n$*) termen. Een (lineair polynomiale)* CHR rangfunctie *definieert de rang van een term of constraint* $f(t_1, \ldots, t_n)$ *als een natuurlijk getal:*

$$\text{rang}(f(t_1, \ldots, t_n)) = a_0^f + a_1^f * \text{rang}(t_1) + \ldots + a_n^f * \text{rang}(t_n)$$

waarbij a_i^f *natuurlijke getallen zijn. Voor elke variabele* X *eisen we dat* $\text{rang}(D) \geq 0$*, voor elke ingebouwde constraint* C *nemen we* $\text{rang}(C) = 0$ *en voor elke CHR-constraint* D *moet* $\text{rang}(D) > 0$*. De rang van een conjunctie is de som van de rangen van de conjuncten:* $\text{rang}((A \wedge B)) = \text{rang}(A) + \text{rang}(B)$*.*

Definitie 3.5 (Rangconditie) *Laat* rang *een CHR rangfunctie zijn. De* rangconditie *van een simplificatieregel* `H <=> G | C ∧ B` *(met* C *een sequentie van ingebouwde constraints en* B *een sequentie van CHR-constraints) is de formule*

$$\forall((G \wedge C) \rightarrow \text{rang}(H) > \text{rang}(B))$$

Definitie 3.6 (Begrensde query) *Een query* G *is* begrensd *als er een (constante) bovengrens is voor de rang van elke instantiatie van* G*.*

Stelling 3.4 (Terminatie en berekeningslengte) *Laat P een CHR-programma zijn dat enkel simplificatieregels bevat. Als voor elke regel in P voldaan is aan de rangconditie dan eindigt P voor elke begrensde query en dan is de* worst-case *berekeningslengte voor zo'n queries de rang ervan.*

Eigen bijdrage. 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 terminatie-bewijzen voor CHR-programma's is een mogelijkheid voor verder onderzoek. Tenslotte dient de toepasbaarheid van technieken uit het domein van herschrijfsystemen onderzocht te worden.

3.2.3 Confluentie

Bestaand werk. Stelling 3.1 houdt in dat het resultaat van een berekening van een gegeven query steeds dezelfde logische betekenis heeft, maar het is niet gegarandeerd dat het resultaat ook syntactisch steeds hetzelfde is. *Confluentie* [1, 7, 6] 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.

Definitie 3.7 (Samenvoegbare toestanden) *Twee toestanden S_1 en S_2 zijn* samenvoegbaar *(joinable) als er toestanden S_1' en S_2' bestaan die gelijk zijn op hernoeming van variabelen na, zodat $S_1 \mapsto^\star S_1'$ en $S_2 \mapsto^\star S_2'$.*

Definitie 3.8 (Confluentie) *Een CHR-programma is* confluent *als voor alle toestanden S, S_1, S_2 geldt dat als $S \mapsto^\star S_1$ en $S \mapsto^\star S_2$ dan zijn S_1 en S_2 samenvoegbaar.*

Stelling 3.5 (Consistentievoorwaarde) *Als P confluent is, dan is $\mathcal{P} \cup CT$ consistent.*

Definitie 3.9 (Kritiek paar) *Als S een niet-triviale direct gemeenschappelijke voorouder-toestand* [2] *is van een simplificatieregel R_1 en een willekeurige regel R_2 met $S \mapsto_{R_1} S_1$ en $S \mapsto_{R_2} S_2$, dan noemen we het koppel (S_1, S_2) het* kritiek paar *van S.*

Stelling 3.6 (Confluentie) *Een CHR-programma zonder oneindige berekeningen is confluent als en slechts als al zijn kritieke paren koppels van samenvoegbare toestanden zijn.*

Eigen bijdrage. Stelling 3.6 geeft een beslisbare, nodig 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 bepaalde niet-samenvoegbare kritieke paren toch niet problematisch zijn. In de praktijk [83] blijkt dat vaak het grootste deel van de niet-samenvoegbare kritieke paren die door de confluentietest gevonden worden, verworpen mogen worden. 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.

[2] *Nontrivial direct common ancestor state* van twee CHR-regels: een minimale toestand waarin beide regels kunnen toegepast worden (zie [1, 36] voor formele definities)

Om een eenvoudig voorbeeld te geven: in veel gevallen zijn een aantal constraints in feite "interne" constraints die niet mogen voorkomen in queries. De confluentietest kan dan detecteren dat bepaalde combinaties van dergelijke interne constraints tot verschillende antwoorden kunnen leiden, al kunnen die combinaties nooit voorkomen als enkel geldige queries worden gegeven. De gebruiker is gewoonlijk niet geïnteresseerd in zo'n problemen die nooit voorkomen. Het is dus zeer wenselijk dat er in een CHR-programma kan opgeven worden welke constraints mogen voorkomen in 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. Als kan aangetoond worden dat een programma confluent is, kan binnen de verfijnde semantiek de volgorde van regeltoepassingen toch naar believen worden aangepast – als er bijvoorbeeld redenen zijn om aan te nemen dat een bepaalde volgorde resulteert in betere efficiëntie.

3.2.4 Vervollediging (completion)

Bestaand werk. Vervollediging is het toevoegen van regels aan een niet-confluent CHR-programma tot het confluent wordt. In [3] 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.

Eigen bijdrage. Een interessant onderwerp om te bestuderen is de relatie tussen vervollediging en gedeeltelijke evaluatie (partial evaluation).

3.2.5 Operationele equivalentie

Definitie 3.10 (Operationele equivalentie) *Twee CHR-programma's P_1 en P_2 zijn* operationeel equivalent *als alle toestanden P_1, P_2-samenvoegbaar zijn, d.w.z. voor elke toestand S waarvoor berekeningen $S \mapsto^\star_{P_1} S_1$ en $S \mapsto^\star_{P_2} S_2$ bestaan met S_1 en S_2 eindtoestanden, zijn S_1 en S_2 varianten van elkaar.*

Bestaand werk. Er is een beslisbare, nodige en voldoende syntactische voorwaarde voor operationele equivalentie van eindigende, confluente CHR-programma's [8].

Eigen bijdrage. 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.

3.2.6 Complexiteitsanalyse

Bestaand werk. In [38, 39] 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:

Stelling 3.7 (Tijdscomplexiteit) *Laat P een CHR-programma zijn dat enkel simplificatieregels bevat. Laat Q een query zijn met worst-case berekeningslengte D. Dan is de worst-case tijdscomplexiteit van een berekening van Q gegeven door*

$$O(D \sum_i (D^{n_i}(O_{H_i} + O_{G_i}) + O_{CB_i}))$$

waarbij de index i over alle simplificatieregels in het programma P gaat, O_H de kost van de matching *van de* head *van een regel voorstelt, O_G de kost van het controleren van de* guard*-conditie, O_{CB} de kost van het toevoegen van de constraints in de* body *en het verwijderen van de CHR-constraints uit de* head*.*

Eigen bijdrage. De bovengrens voor de tijdscomplexiteit zoals gegeven wordt door stelling 3.7 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.

3.3 Optimalisaties

3.3.1 Bestaand werk

We sommen nu enkele optimalisaties op, waarvan sommige reeds geïmplementeerd zijn in bestaande CHR-versies. In [52, 31] 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 (set semantics). 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 ook 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).

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.

Verwijderen van redundante regels. Als uit een programma P een regel R geschrapt wordt en het resulterend programma operationeel equivalent is met P, dan is R een overbodige regel die niet gecompileerd moet worden [13]. In [4] wordt een gespecialiseerde operationele equivalentietest geschetst om dit soort redundante regels te detecteren.

Delay avoidance aan de hand van antimonotoniciteit [73] is een optimalisatie die ontwikkeld werd door onze onderzoekseenheid. Er wordt geprobeerd het aantal variabelen die een constraint kunnen triggeren te reduceren – hetgeen op zich al een zekere overhead elimineert – zodat zo weinig mogelijk overbodige 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.

3.3.2 Eigen bijdrage

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.

Bij het zoeken naar *never stored* constraints worden bijvoorbeeld in huidige implementaties enkel eenvoudige gevallen zonder *guard* en waarbij de argumenten van de constraint vrije variabelen zijn gedetecteerd. Een verfijning hiervan kan ontwikkeld worden, die de constraint `p` als *never stored* kan detecteren in programma's zoals

```
p(N) <=> N >= 0 | B
p(N) <=> N < 0 | B
```

waar de *guard*-condities alle mogelijke gevallen omvatten, of in programma's zoals

```
p([]) <=> B
p([X|R]) <=> B
```

waarbij de argumenten van de constraints alle mogelijke gevallen omvatten (indien geweten is dat het argument van `p/1` van het type *list* is kan zelfs nog iets meer geoptimaliseerd worden). In de volgende sectie bespreken we een framework voor abstracte interpretatie, waarbinnen we deze en andere analyses/optimalisaties zullen formuleren en er op die manier meer inzicht in verwerven.

3.4 Abstracte interpretatie

Abstracte interpretatie (AI) [21, 15, 16, 14, 63] 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.

3.4.1 Bestaand werk

Bestaande AI-frameworks – zoals voor Prolog [15, 14] – 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 [30] 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.

3.4.2 Eigen bijdrage

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. Ook het gebruik van type-informatie om analyses te verbeteren dient onderzocht te worden – zoals in het voorbeeldje uit sectie 3.3.2.

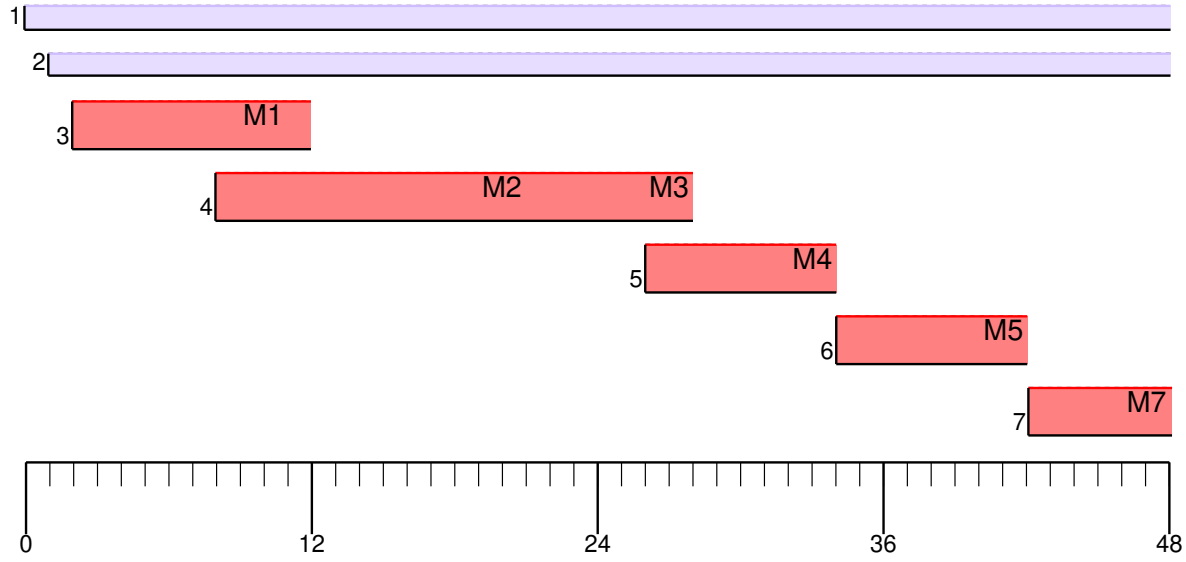
3.5 Implementatie van CHR

3.5.1 Bestaand werk

De eerste CHR-implementatie, in de zomer van 1991, was een interpreter geschreven in ECL^{i}PS^e Prolog genaamd Cheer [34, 43]. Sindsdien zijn er implementaties van CHR in verschillende *host*-talen: **Common LISP** [50], **ECL^{i}PS^e** [42] (als library), **Sicstus Prolog** [54, 53, 58, 56, 55], **YAP Prolog** [22], **Java** [9, 11, 86], **HAL** [26, 52, 24, 31], **Haskell** (Chameleon,TIE) [47, 80], **XSB** [77, 76, 75], **hProlog** en **SWI-Prolog** [75]. Deze implementaties zijn allen gebaseerd op compilatie naar de *host*-taal.

3.5.2 Eigen bijdrage

Er zal gewerkt worden aan de integratie van analyses/optimalisaties in de *bootstrapped* CHR-compiler, die als gemeenschappelijke compiler voor verschillende platformen dienst zal doen. Op die manier worden deze optimalisaties/analyses onmiddellijk beschikbaar voor elk platform en niet slechts voor één specifiek platform. Dit zal een fundamentele vooruitgang zijn in het CHR-onderzoek. In de loop van dit project zullen verder specifiek verfijningen aangebracht worden aan de bestaande CHR-implementaties in **HAL** en in **Java**, voornamelijk wat betreft de gebruikte datastructuren en algoritmes. Zo zal de efficiëntie van *state-of-the-art* Prolog-implementaties ook in Java beschikbaar worden.



Figuur 1: Planning van het onderzoeksproject

4 Planning

1. **Literatuurstudie** (3 maanden)
   De eerste periode bestaat uit een 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** (6 maanden)
   De volgende periode van het onderzoek is een gedetailleerde studie van de bestaande CHR-implementaties en analyse- en optimalisatietechnieken die daarin gebruikt worden. Ook dit onderdeel wordt gespreid over de vier jaren.

3. **Ontwikkelen Framework voor Abstract Interpretation** (9 maanden)
   In deze fase wordt een theoretisch kader ontwikkeld voor abstracte interpretatie van CHR-programma's, waarbinnen geredeneerd kan worden over en correctheid kan bewezen worden van de analyses en optimalisaties uit de volgende fase. De ontwikkeling van dit framework resulteert in de verwezenlijking van de eerste mijlpaal (M1). We hopen tot een dieper inzicht te komen in bestaande optimalisaties door deze te bestuderen binnen dit framework.

4. **Ontwerpen en implementeren van Analyses/Optimalisaties** (12 maanden)
   Het hoofdonderzoek van dit project bestaat uit het ontwerpen en implementeren van verschillende nieuwe analyses en optimalisaties binnen het framework voor abstracte interpretatie. Dit resulteert in de tweede mijlpaal (M2). Eveneens zal een belangrijk deel van deze periode besteed worden aan verificatie en evaluatie van de ontwikkelde optimalisaties: enerzijds zullen we de correctheid ervan moeten aantonen, anderzijds zullen we onder andere door middel van *benchmarks* en experimenten met programma's op *real-life*-schaal de resulterende efficiëntieverhogingen testen. Dit resulteert in de derde mijlpaal (M3).

5. **Bootstrapped CHR-compiler** (6 maanden)
   In dit volgend stadium wordt gewerkt aan de integratie van de ontwikkelde analyses in de *bootstrapped* CHR-compiler. De verwezenlijking hiervan resulteert in de vierde mijlpaal (M4).

6. **Java-implementatie** (6 maanden)
   In deze fase wordt de Java-implementatie van CHR verder ontwikkeld, steunend op het vorige eigen werk en het bestaand werk hieromtrend. Dit resulteert in de vijfde mijlpaal (M5).

7. **Schrijven van het proefschrift** (6 maanden)
   Tenslotte worden 6 maanden voorzien voor het schrijven van het proefschrift, hetgeen resulteert in de laatste mijlpaal (M6).

In de praktijk zullen deze verschillende fases, die hier voor de eenvoud min of meer sequentieel werden voorgesteld, gedeeltelijk overlappen voor zover onderlinge afhankelijkheden dat toelaten.

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)* [5, 41] 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, ECL^{i}PS^e Prolog.

- *Het opstellen van lessenroosters* is een typisch *scheduling*-probleem dat met Constraint Logic Programming kan opgelost worden. In [10] 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* [44] 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* [46] 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 [74] is een referentie-implementatie voor Concurrent Constraint-based Memory (CCM) machines [70], een framework voor Java geheugenmodellen. JmmSolve werd ontwikkeld door onze onderzoeksgroep.

Andere domeinen waarin CHR nuttig is gebleken zijn onder andere multi-agent systemen [78], financiële informatica [33], computationele linguistiek [67, 69] en cognitieve robotica [82, 81].

Referenties

[1] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 252–266, Schloss Hagenberg, Austria, 1997. Springer LNCS.

[2] Slim Abdennadher. A Language for Experimenting with Declarative Paradigms, Second Workshop on Rule-Based Constraint Reasoning and Programming. In *6th Intl Conf on Principles and Practice of Constraint Programming (CP'2000)*, Singapore, 2000.

[3] Slim Abdennadher and Thom Frühwirth. On completion of constraint handling rules. In *Fourth International Conference on Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, October 1998.

[4] Slim Abdennadher and Thom Frühwirth. Integration and Optimization of Rule-based Constraint Solvers. In M. Bruynooghe, editor, *Logic Based Program Synthesis and Transformation (LOPSTR 2003), Revised Selected Papers*. LNCS, Springer Verlag, 2004. to appear.

[5] Slim Abdennadher, Thom Frühwirth, and Peter Blenninger. Rent Estimates with Constraints over the Internet – System Description. In *Journées Francophones de Programmation en Logique et programmation par Contraintes (JFPLC'96)*, Clermont Ferrand, France, June 1996.

[6] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On confluence of constraint handling rules. In *Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, LNCS 1118. Springer, August 1996.

[7] Slim Abdennadher, Thom Fruhwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the Second International Conference on Principles and Practice of Constraint Programming*, 4(2), May 1999.

[8] Slim Abdennadher and Thom Früwirth. Operational Equivalence of CHR Programs and Constraints. In *Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*. Springer LNCS, Alexandra, Virginia, October 1999.

[9] Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java Constraint Kit. *Electronic Notes in Theoretical Computer Science*, 64, 2000.

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

[11] Slim Abdennadher and Matthias Saft. A Visualization Tool for Constraint Handling Rules. In *11th Workshop on Logic Programming Environments*, Paphos, Cyprus, 2001.

[12] Frédéric Benhamou and William J. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. Technical report, Bell Northern Research, 1992.

[13] Sebastian Brand. A Note on Redundant Rules in Rule-based Constraint Programming. In *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 109–120, 2002.

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

[15] 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.

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

[17] Michael Codish and Bart Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proceedings of Static Analysis, Aachen, Germany*, SAS'94, LNCS Vol. 864, pages 281–296, 1994.

[18] Michael Codish and Bart Demoen. Analysing logic programs using "prop"-ositional logic programs and a magic wand. *Journal of Logic Programming*, 25(3):249–274, dec 1995.

[19] 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, jan 1995.

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

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

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

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

[24] María García de la Banda, David Jeffery, Kim Marriott, Peter J. Stuckey, Nicholas Nethercote, and Christian Holzbaur. Building constraint solvers with HAL. In *Proceedings of the Seventeenth International Conference on Logic Programming (ICLP'2001)*, pages 90–104. LNCS, Springer Verlag, December 2001.

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

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

[27] 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.

[28] Marc Denecker. Semantic abstract interpretation. In *Proceedings of Workshop on Abstract Interpretation of Logic Languages*, Eilat, Israel, 1995.

[29] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[30] Gregory J. Duck, María García de la Banda, Tom Schrijvers, and Peter J. Stuckey. Abstract Interpretation for Constraint Handling Rules. Technical Report CW 391, K.U.Leuven, Departement of Computer Science, 2004. to appear.

[31] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending Arbitrary Solvers with Constraint Handling Rules. In D. Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, 2003.

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

[33] Aykut Firat, Stuart Madnick, and Benjamin Grosof. Financial Information Integration in the Presence of Equational Ontological Conflicts. In *No 4408-02, Working papers from Massachusetts Institute of Technology (MIT), Sloan School of Management*, 2002.

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

[35] 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, March 1995.

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

[37] Thom Frühwirth. Proving termination of constraint solver programs. In Krzysztof R. Apt, Antonis C. Kakas, Eric Monfroy, and Francesca Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus*, volume 1865 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, May 2000.

[38] Thom Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In Dieter Fensel, Fausto Giunchiglia, Deborah McGuinness, and Mary-Anne Williams, editors, *KR2002: Principles of Knowledge Representation and Reasoning*, pages 547–557. Morgan Kaufmann, San Francisco, California, 2002.

[39] Thom Frühwirth. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. In Alessandra Di Pierro and Herbert Wiklicky, editors, *QAPL'01 - Quantitative Aspects of Programming Languages, Selected Papers*, volume 59(3) of *Electronic Notes in Theoretical Computer Science (ENTCS)*. 2002.

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

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

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

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

[44] Thom Frühwirth and Pascal Brisset. Optimal Placement of Base Stations in Wireless Indoor Telecommunication. In *Fourth International Conference on Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, October 1998. Winner of telecom application contest.

[45] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint logic programming - an informal introduction. In *Logic Programming in Action*. Springer LNCS 636, September 1992.

[46] Joost Geurts, Jacco van Ossenbruggen, and Lynda Hardman. Application-specific constraints for multimedia presentation generation. In *Proceedings of the International Conference on Multimedia Modeling 2001 (MMM01)*, pages 247–266, CWI, Amsterdam, The Netherlands, November 2001.

[47] Kevin Glynn, Peter J. Stuckey, and Martin Sulzmann. A general type class framework. Technical Report TR2001/26, Department of Computer Science, The University of Melbourne, 2001.

[48] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

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

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

[51] Christian Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.

[52] Christian Holzbaur, María García de la Banda, David Jeffery, and Peter J. Stuckey. Optimizing compilation of constraint handling rules. In P. Codognet, editor, *17th International Conference on Logic Programming (ICLP)*, pages 74–89. LNCS, Springer-Verlag, 2001.

[53] Christian Holzbaur and Thom Frühwirth. Compiling Constraint Handling Rules. In *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, 1998.

[54] Christian Holzbaur and Thom Frühwirth. Constraint Handling Rules Reference Manual, Release 2.2. Technical Report TR-98-01, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, Wien, 1998.

[55] Christian Holzbaur and Thom Frühwirth. A Compiler for Constraint Handling Rules based on Partial Evaluation. In *9th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, Venice, Italy, September 1999.

[56] Christian Holzbaur and Thom Frühwirth. Compiling Constraint Handling Rules into Prolog with Attributed Variables. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Paris, France, September/October 1999.

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

[58] Christian Holzbaur and Thom Früwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), april 2000.

[59] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM Press, 1987.

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

[61] Joxan Jaffar, Michael J. Maher, Kim Marriot, and Peter J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, October 1998.

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

[63] 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.

[64] Gerda Janssens, Maurice Bruynooghe, and Veroniek Dumortier. A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In *Proceedings of the ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, pages 18–35, jun 1995.

[65] 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 Constrauction*, pages 172–187, apr 1994.

[66] Antonis C. Kakas, Bert Van Nuffelen, and Marc Denecker. A-system : Problem solving through abduction. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, volume 1, pages 591–596. IJCAI, inc and AAAI, Morgan Kaufmann Publishers, Inc, 2001.

[67] Frank Morawietz and Philippe Blache. Parsing Natural Languages with CHR. draft, 2002.

[68] Bert Van Nuffelen and Antonis C. Kakas. A-System : Programming with abduction. In *Logic Programming and Nonmonotonic Reasoning, LPNMR 2001, Proceedings*, volume 2173 of *LNAI*, pages 393–396. Springer Verlag, 2001.

[69] Gerald Penn. Applying Constraint Handling Rules to HPSG. In *Proceedings of the First International Conference on Computational Logic (CL2000), Workshop on Rule-Based Constraint Reasoning and Programming*, UK, August 2000. Imperial College.

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

[71] Danny De Schreye and Christophe Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Gen. Comput.*, 13(2):117–154, 1995.

[72] Tom Schrijvers. Benchmarks used for comparing CHR systems on different Prolog systems, 2004. Available at *http://www.cs.kuleuven.ac.be/~toms/Research/chr.html*.

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

[74] Tom Schrijvers and Bart Demoen. JmmSolve: a generative reference implementation of CCM machines. Technical Report CW 379, K.U.Leuven, Department of Computer Science, 2004.

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

[76] Tom Schrijvers and David S. Warren. Constraint Handling Rules and Tabled Execution. In *20th International Conference on Logic Programming (ICLP'04)*, Saint-Malo, France, September 2004.

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

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

[79] Greg Sidebottom. *A Language for Optimizing Constraint Propagation.* PhD thesis, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada, November 1993.

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

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

[82] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 2004. *to appear.*

[83] Thom Früwirth Tom Schrijvers. Implementing and Analysing Union-Find in CHR. Technical Report CW 389, K.U.Leuven, Department of Computer Science, July 2004.

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

[85] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

[86] Armin Wolf. Adaptive Constraint Handling with CHR in Java. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 256–270. Springer-Verlag, 2001.