Journal of Universal Computer Science, vol. 13, no. 6 (2007), 701-720
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © *J.UCS*

Constraint Programming Architectures: Review and a New Proposal

Jacques Robin
(Centro de Informática – Universidade Federal de Pernambuco (CIn-UFPE), Brazil
robin.jacques@gmail.com)

Jairson Vitorino
(Centro de Informática – Universidade Federal de Pernambuco (CIn-UFPE), Brazil
jairson@gmail.com)

Armin Wolf
(Fraunhofer Institut - Rechnerarchitektur und Softwaretechnik (FIRST), Germany
Armin.Wolf@first.fraunhofer.de)

Abstract: Most automated reasoning tasks with practical applications can be automatically reformulated into a constraint solving task. A constraint programming platform can thus act as a unique, underlying engine to be reused for multiple automated reasoning tasks in intelligent agents and systems. We identify six key requirements for such platform: expressive task modeling language, rapid solving method customization and combination, adaptive solving method, user-friendly solution explanation, efficient execution, and seamless integration within larger systems and practical applications. We then propose a novel, model-driven, component and rule-based architecture for such a platform that better satisfies as a whole this set of requirements than those of currently available platforms.

Keywords: constraint programming, model-driven architecture, software reuse
Categories: D.1.6, D.2.11, D.2.13, I.2.3, I.2.4, I.2.5, I.2.8

1 Introduction

Over the last two decades, the practical inference needs of intelligent agents and systems have led the field of automated reasoning to vastly diversify from its roots in monotonic deduction. It now also encompasses abduction, default reasoning, inheritance, belief revision, belief update, planning, constraint solving, optimization, induction and analogy. Many specialized methods are now available to efficiently implement specific subclasses of these tasks in conjunction with some specific knowledge representation languages. These languages vary in terms of their two commitments, epistemological (*e.g.*, logical, plausibilistic, or probabilistic) and ontological (*e.g.*, propositional, first-order relational, first-order object-oriented, high-order relational or high-order object-oriented) [Russell and Norvig, 03]. This diversity in tasks, methods and languages inhibits pervasive embedding of automated reasoning functionalities in applications. It confuses the development teams of most applications who generally have a sparse background in automated reasoning and it seems to prevent cost-cutting reuse of a single generic platform for many such purposes. However, a way out of this dilemma is suggested by the fact that whether using

propositional [WebCHR 07]), first-order relational [WebCHR 07]) or object-oriented [Ait-Kaci and Nasr, 86] representations with either logical [Frühwirth, 94], plausibilistic [Jim and Thielscher, 06] or probabilistic [Costa et al, 03] semantics, the reasoning tasks of deduction [WebCHR 07]), abduction [Abdennadher S., 01], default reasoning [Abdennadher S., 01], inheritance [Ait-Kaci and Nasr, 86.], belief revision [Jim and Thielscher, 06], belief update [Thielscher, 02] and planning [Thielscher, 02] have now all being reformulated as special cases of constraint solving. An adequate **Constraint Programming Platform** (CPP) could thus be successfully reused as an all subsuming engine to implement and seamlessly integrate those diverse tasks, methods, and languages [Robin and Vitorino, 06].

In this paper, after providing some background on constraint programming (Section 2), we first identify the key requirements of CPP platforms (Section 3). We then review the software architectures of the current CPP and evaluate them with respect to those requirements (Section 4). We then propose a new CPP architecture based on component, aspect and object models, as well as rewrite rules for constraint handling and model transformation, and we argue that it better fulfils the CPP requirements than current CPP architectures (Section 5). We conclude by quickly describing the current status and next steps in our implementation of this architecture (Section 6).

2 Constraint Programming

Constraint programming is the cutting-edge IT to automate and optimize tasks such as resource allocation, scheduling, routing, layout and design verification in the most diverse industries. A **constraint program** models an application domain using a subset of first-order logic restricted to atom conjunctions. Each model consists of

- a set of variables X_1, ..., X_n;
- for each variable X_i, a corresponding domain D_i of possible values (*i.e.*, constant symbols), *e.g.*, {red, green, blue} or floating point numbers;
- for each domain D_i, a set of functions f_i^1, ... f_i^o with domain and range in D_i (*e.g.*, mix, +, **);
- a logical formula F of the form $C_1 \wedge ... \wedge C_p$ where each C_i is an atom that relates terms formed from variables, constants and functions (*e.g.*, $X_1 \neq X_2$, X_1 = blue, X_1 = mix(red,blue), $X_1 \geq 2.0000 * 3.1416 * X_2$).

Each C_i in F is called a **constraint** because it restricts the possible value combinations of the variables X_1, ... X_n that occur in it within their respective domains D_1, ... D_n. A solver takes as input a constraint problem instance P_i in the form of a conjunctive formula F_i. If P_i is exactly constrained (*e.g.*, $X,Y,Z \in N \wedge X+Y=Z \wedge 1<X \wedge X<Z \wedge X<Y \wedge Y<Z \wedge Z<6$) the solver returns as output a formula F_o in determined solved form (*i.e.*, of the form $X_1 = v_1 \wedge ... \wedge X_n = v_n$ with X_is variables and v_is constants) that is logically equivalent to F_i (*e.g.*, $X=2 \wedge Y=3 \wedge Z=5$)**.** If P_i is overconstrained, (*e.g.*, $X,Y,Z \in N \wedge X+Y=Z \wedge 1<X \wedge X<Z \wedge X<Y \wedge Y<Z \wedge Z<4$) the solver returns `false`. If P_i is underconstrained (*e.g.*, $X,Y,Z \in N \wedge X+Y=Z \wedge 1<X \wedge X<Z \wedge X<Y \wedge Y<Z \wedge Z<7$) the solver returns a formula F_o logically equivalent to F_i but syntactically *simpler* (*e.g.*, ($X=2 \wedge 3 \leq Y \wedge Y \leq 4 \wedge 5 \leq Z \wedge Z \leq 6 \wedge Z = Y + 2$), or

(X=2 ∧ ((Y=3 ∧ Z=5) ∨ Y=4 ∧ Z=6). Key simplicity factors include fewer constraints, fewer variables, shorter atoms, higher proportion of atoms in determined solved form or (undetermined) solved form (*i.e.*, of the form $X_1 = t_1 \wedge ... \wedge X_n = t_n$ where each X_i is a variable and each t_i a term not containing occurrences of X_1, ..., X_{i-1}, X_{i+1}, ..., X_n). Figure 1 gives a simple illustrative example of a constraint solving task and solution. It is an instance of the classic map coloring problem: how to allocate variables representing regions on a map to values from a finite domain of colors, such that any two neighboring regions are allocated different colors?

Solving Task as Logic Formula:

R1 ∈ {r,b,g} ∧ R2 ∈ {r,b,g} ∧ R3 ∈ {r,b,g} ∧
R4 ∈ {r,b,g} ∧ R5 ∈ {r,b,g} ∧ R6 ∈ {r,b,g} ∧
R7 ∈ {r,b,g} ∧ R1 ≠ R2 ∧ R1 ≠ R3 ∧
R1 ≠ R4 ∧ R1 ≠ R7 ∧ R2 ≠ R6 ∧ R3 ≠ R7 ∧
R4 ≠ R5 ∧ R4 ≠ R7 ∧ R5 ≠ R6 ∧ R5 ≠ R7

One Solving Solution as Logic Formula:

R1=g ∧ R2=b ∧ R3=r ∧ R4=r ∧ R5=g ∧ R6=r ∧ R7=b

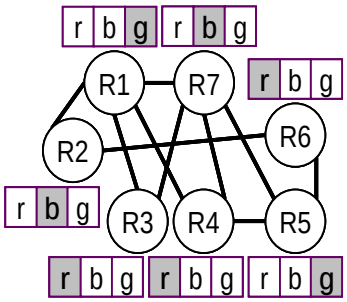

Figure 1: Constraint Solving Task and Solution Example

Figure 1 shows the problem and one solution for it in two formulations: a logic formulation on the left hand side, and a constraint graph formulation on the right-hand side. This graph contains one node per variable/region and one arc per neighboring constraint. Above or below each node, the corresponding variable's domain is shown (with `r, b,` and `g` respectively abbreviating `red, blue` and `green`). One color allocation solution is indicated by the grey box in each domain.

A CPP integrates the constraint task modeling syntax with that of a general purpose host programming language. It typically works by interleaving three main subtasks:

- **simplifying** a set of constraint into a simpler equivalent, *e.g.*, simplifying R ∈ {r,b} ∧ R = g into false, or simplifying R ∈ {r,b,g} ∧ R ∈ {r,b} into R ∈ {r,b};
- **propagating** some logical consequences of a set of constraint thus making explicit some constraints that they entail, *e.g.*, propagating R1 ∈ {r,b,g} ∧ R1 ≠ R2 ∧ R2 = r ∧ R1 ≠ R3 ∧ R2 = b into R1 = g;
- **searching** the space of possible variable combination values, *e.g.*, searching in R1 ∈ {r,b} ∧ R2 ∈ {r,b} given R1 ≠ R2 to find (R1 = r ∧ R2 = b) ∨ (R1 = b ∧ R2 = r).

3 Constraint Programming Platforms Requirements

A thoughtful engineering process chooses between various architectural designs based on a prior identification of the key functional and non-functional requirements to make the software under construction a most practical and useful tool. What are such requirements for a CPP?

The first requirement is to provide an **expressive input task modeling language**. This is clearly decisive for the CPP's versatility and its ability to fulfill the promise of an integrated platform for the diverse automated reasoning needs of an application. Less obvious, is that it is also crucial for its overall efficiency, for the key of efficient solving often lays as much on the way the task in modeled as it does on the chosen solving method and how this method is implemented. A platform with a very limited task modeling language does not allow for many logically equivalent formulations of the same task. For many tasks, it thus risks of not supporting any of the formulations that lend themselves to efficient solving.

For the same versatility and overall efficiency reasons, the second key requirement is to provide many solving **method customization and combination facilities**. This allows exploiting the peculiarities of subtly different solving task sub-classes, which often results in dramatic efficiency gains for many task instances.

A third requirement in the overall efficiency puzzle is **efficient implementation** techniques for the available solving methods. It is somewhat conflicting with the first two since raising expressiveness often has the undesirable side-effect of increasing theoretical worst-case complexity, while method tweaking, mixing and matching facilities often brings some configuration and assembly run-time overhead.

A fourth requirement is **solution adaptation**. Consider two task instances T_1 and T_2 that differ only by a few more and/or a few less constraints. Once it has computed a solution S_i^1 for T_1, an adaptive solver is able to reuse S_i^1 so as to (a) find a solution S_a^2 for T_2 that is minimally distant from S_i^1 and (b) find it more quickly than a solution S_s^2 computed from scratch. A non-adaptive solver can only solve T_2 from scratch which may lead to a solution S_s^2 that has very few common components with S_i^1. In the practical application context, a large distance between S_s^2 and S_i^1 often makes S_s^2 unusable. For example, imagine than S_i^1 was an initial task schedule for a large engineering project that requires adjustment due to execution delays. A schedule S_s^2 computed from scratch for the uncompleted tasks may well promise a shorter revised delivery date estimate than that of an adaptive schedule S_a^2 which additionally strives for stability from the original S_i^1. But if S_s^2 allocates resources in a vastly different way than S_i^1 for the uncompleted tasks, the associated allocation reshuffling overhead cost generally far outweigh the gains of a shorter delivery date. Most available CPP do not provide adaptive solving.

The fifth key requirement of a practical CPP is a user-friendly, detailed **solution explanation** facility. Consider again the examples above, where a CPP propose alternative rescheduling plans for a late several billion dollars engineering project of a company flagship product. With so much at stake, adopting one of these alternatives requires the CPP to provide explanations that justify its discarding millions of possible others. It also requires to provide a concise summary of the contrasting trade-offs embodied in two proposed alternative plans. The key challenge for such explanations and summaries is to be as directly understandable as possible by the

executives with decision making power but no technical constraint solving background.

The sixth key requirement for a practical CPP is **seamless integration** within a variety of applications. Today, most of them are developed using **Component-based Object-oriented Imperative Platform** (COIP) such as EJB or .Net. There are two main reasons for this. The first is that these frameworks provide as consolidated built-ins the application independent services that constitute most of the running code of most information systems. These built-ins include high level API for database and GUI development, transparent secure persistence, scalable concurrent transaction handling and distributed deployment (including fully automated code generation to publish the various system functionalities as web services). The second is that the most advanced full life cycle software engineering methods and supporting CASE tools are based on the COI paradigm. This allows full integration of these CASE tools with COIP IDE and brings high gains in software productivity and quality. Thus, in today's practice, seamless integration of a CPP in applications, means encapsulating it as a Java or .Net component.

4 Constraint Programming Platform Architectures

Having defined the requirement of practical CPP, let us now review the various software architectures of available CPP and evaluate how they meet each of these requirements.

4.1 Component-based Object-oriented Imperative Platform API

The simplest CPP architecture is a library of classes or a component framework in a COIP, as summarized in Figure 2.

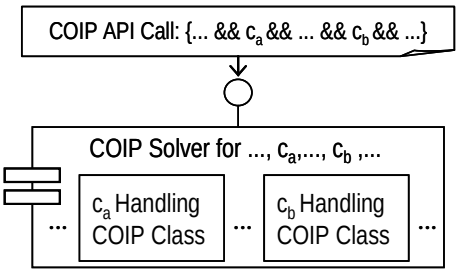

Figure 2: COIP API CPP

This is the case for example of the Java firstcs library [Wolf, 06] and the commercial C++ ILOG Solver [Puget, 94]. CP with such COIP consists in writing an object-oriented program that uses the CP API to first create variables in the domains built in the API, then create constraints among these variables for which solvers are built in the API and finally call the primitive operations for constraint propagation and finite domain combinatorial search. These primitive operations process the variable and

constraint objects created by the previous calls which together defined the constraint solving task. They return a collection of solution objects for that task.

The first weak point of this architecture is to provide the least expressive constraint task modeling language: a restriction of the one described in Section 2, with the constraints, functions and constants all from the *closed, fixed set* implemented by the class library. Another drawback of the COIP architecture is its poor customizing facility, which is possible only through time-consuming low-level imperative code alterations. A COIP API provides some low-cost method combination facilities through component assembly. But the generation of solving step justifications for adaptation and explanations are cross-cutting concerns that cannot be encapsulated as separate components. To the best of our knowledge, there is neither an aspect-oriented nor comprehensively adaptive COIP API available to date. The main strengths of the COIP API architecture are implementation efficiency for the fixed set of available methods, seamless constraint solving services integration in application as components or classes, and the availability of COIP IDE and GUI development API.

The COIP API architecture

- represents the solving tasks procedurally by embedding API operation calls inside a special purpose COIP program;
- structures the solving methods declaratively as components and objects, but represents its behavioral details procedurally as operations;
- deploys the code as compiled components and objects;
- uses a compiling method that is generally structured declaratively as objects, but represented in details procedurally as operations.

4.2 Prolog + Procedural Libraries

The most widely used CPP architecture is the so-called parametric **Constraint Logic Programming** scheme CLP(D_1,...,D_q) [Marriott and Stuckey, 98]. The task modeling language of CLP extends that of Section 2 with the equivalence and disjunction connectives and with arbitrary symbolic atoms. A CLP(D_1, ..., D_n) program consists of Horn rules of the form: H :- G_1 ,..., G_r. The **head** H is an *arbitrary* first-order symbolic atom, and each goal $\mathbf{G_i}$ is either a symbolic atom (which appears as head in another rule) or a constraint atom (which does *not* appear as head in another rule) from a *fixed set* of built-ins that relate terms formed from functions and constants of a given domain D_i. The goal conjunction is called the **body** of the rule. Under CLP's closed-world assumption [Russell and Norvig, 03], the logical semantics of each rule subset sharing the same head, {H :- G_1^1,..., G_r^1., ..., H :- G_1^s,..., G_t^s.}, is $H \Leftrightarrow (G_1^1 \wedge ... \wedge G_r^1) \vee ... \vee (G_1^s \wedge ... \wedge G_t^s)$. The overall CLP program semantics is the conjunction of these equivalences. This extended expressivity allows modeling complex domain knowledge with high-level concepts declaratively defined from built-in constraints using Horn rules. Since these rules can be recursive and can contain function symbols, CLP provides a Turing-complete declarative constraint modeling language. It also provides a constraint query language: queries are simply headless rules. In CLP parlance, the task models and queries that a COIP API permits to express correspond to headless rules containing only built-in constraint atoms.

The first internal CPP architecture following the CLP(D_1,...,D_q) scheme is shown in Figure 3.

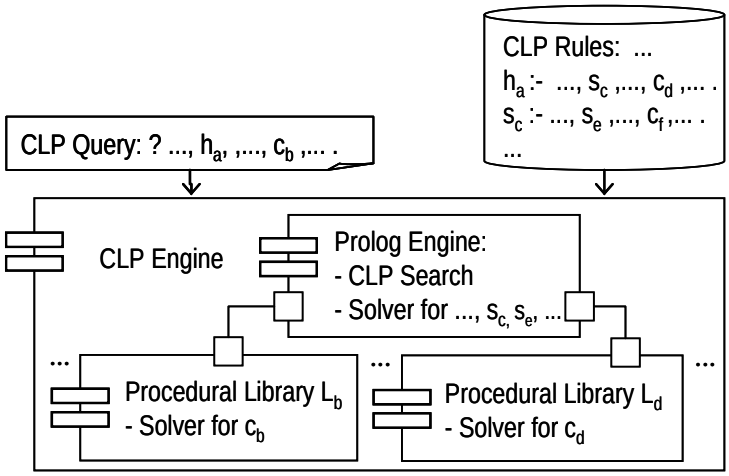

Figure 3: Prolog with Procedural Solver Libraries CPP.

This architecture consists of n+1 components: a Prolog engine and n specialized library, L_1, ..., L_q, one for each domain D_i. Each library implements in a low-level imperative language (generally C or C++) fixed constraint simplification and propagation algorithms and heuristics, fine-tuned to the built-in constraints over D_i terms. The Prolog engine provides the solver for the arbitrary symbolic atoms, as well as a generic **Chronological Backtracking** (CBT) to search valid value combinations from underconstrained finite domains not reducible to singletons through simplification and propagation. Most CLP engines are simple extensions of backward chaining Prolog engines that upon encountering a constraint goal C_i of domain Di adds it to a constraint store of the form $C_1 \wedge ... \wedge C_u$ and calls the imperative library for D_i to solve the updated store. If the store simplifies to `false`, the engine backtracks to the previous goal. Otherwise, it proceeds to the next goal. Data is exchanged between the solving library and the Prolog engine through instantiations of logical variables shared among the rule-defined goals and the built-in constraint goals.

Today, most Prolog platforms include some set of procedural libraries for CLP. This is the case in particular of SWI Prolog [SWI 07], XSB Prolog [XSB 07], YAP Prolog [YAP 07], SICStus Prolog [SICStus 07], as well as the CHIP [CHIP, 07] ECLiPSe [ECLiPSe 07] CLP platforms. Beyond task model expressiveness, the other strong point of the CLP Scheme is its efficient implementation, combining special purpose procedures with general purpose compiled Prolog code. As for weak points, the first is method customization and combination that requires changing low-level imperative library code that is not component-based and often not even object-oriented. The other weaknesses of the CLP scheme are inherited from Prolog: (a) no adaptation, (b) verbose, reasoning trace rarely presented in user-friendly GUIs supporting browsing at multiple abstraction levels, and (c) extremely difficult

integration within applications for lack of components, interfaces, encapsulation, concurrent execution, and standard API for databases and GUI. More often than not, CLP engines only offer brittle bridges to C, C++ or more rarely Java as sole mean of integration. The wide, conceptual impedance mismatch between the logic programming and COI paradigms constitutes in itself a serious integration barrier.

The Prolog + Library architecture

- represents the solving task declaratively by CLP Horn rules;
- represents the solving method in part declaratively as CLP Horn rule and in part procedurally as imperative libraries;
- deploys the code as a rules to be applied by a CLP engine which is accessible via programming language bridges with no access to the embedded procedural solvers;
- uses a compiling method that are declarative as Horn rules for the Prolog engine but procedural for the constraint libraries.

4.3 Prolog + CHR

Constraint Handling Rules (CHR) [Frühwirth, 94] was initially conceived to bring rapid method customization and combination to the CLP scheme by making it fully rule-based. The idea is to substitute by a CHR base each procedural built-in solver of a CLP engine. The task modeling language of Prolog + CHR solver remains the same than in the Prolog + Library approach, since CHR are only used to declaratively define solving *methods* and not tasks. A CHR program is a set of constraint simplification rules, which are conditional rewrite rules of the form, S_1 ,..., S_a <=> G_1 ,.., G_b | B_1 ,..., B_c, and a set of constraint propagation rules, which are guarded production rules of the form: P_1 ,..., P_d ==> G_1 ,.., G_e | B_1 ,..., B_f. Each S_i, P_i, G_i and B_i is a constraint atom. The S_is and P_is are called **heads**, the G_is **guards** and the B_is **goals**. A goal conjunction is called a **body**. Constraint atoms that appear in a CHR head can only appear in other CHR heads and in CLP rule goals. They are called **Rule Defined Constraint** (RDC) atoms to distinguish them from **Built-In Constraint** (BIC) atoms that can appear only as guards and goals in CHR and only as head of Prolog rules that contain no constraint atoms in their bodies. Both kinds of constraint atoms can appear in CHR and CLP rule bodies. The logical semantics of simplification and propagation rules are $G_1 \wedge ... \wedge G_b \Rightarrow (S_1 \wedge ... \wedge S_a \Leftrightarrow B_1 \wedge ... \wedge B_c)$ and $G_1 \wedge ... \wedge G_e \Rightarrow (P_1 \wedge ... \wedge P_d \Rightarrow B_1 \wedge ... \wedge B_f)$ respectively. A CHR engine maintains a constraint store. Operationally, a rule fires when all its heads match against some RDC in the store, while its guards (together with the logical variable bindings resulting from the match) are entailed by the BIC in the store. A fired CHR rule adds its goals to the store. In addition, a fired simplification rules also deletes its heads from the store. A Prolog + CHR engine proceeds as a Prolog + Library engine, except that constraints are solved by forward chaining CHR rules instead of by calling a library procedure. CHR forward chaining stops when the store simplifies to `false` or when it reaches a fixed point, *i.e.*, when no applicable rules can further simplify the store nor add new constraints to it.

As shown in Figure 4, Prolog plays multiple roles in the Prolog + CHR CPP architecture.

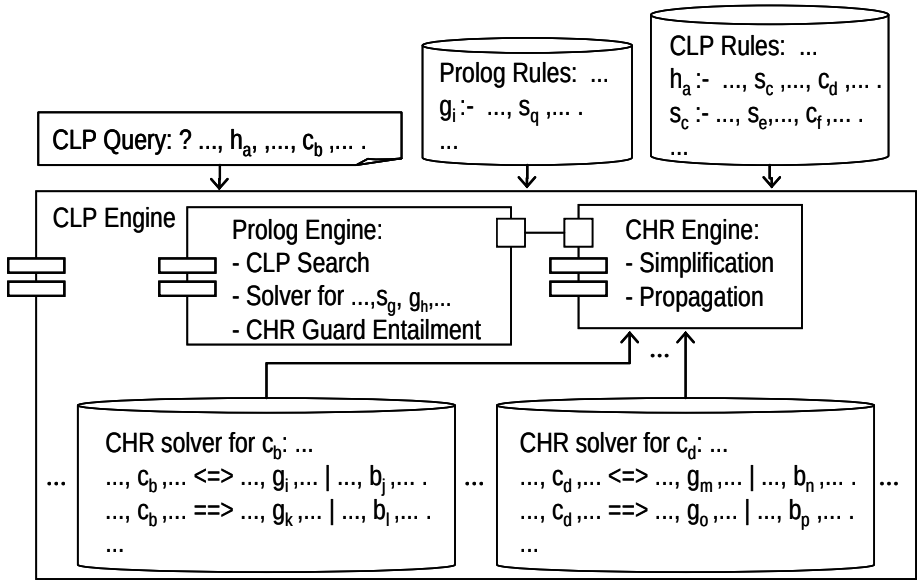

Figure 4: Prolog with CHR CPP Architecture.

First it solves the arbitrary symbolic constraints of the CLP model. Second, its built-in CBT search is reused to search finite domains that cannot be entirely reduced through CHR simplifications and propagations. Third, it is used as host platform to implement the CHR built-in constraints. Fourth, its built-in unification is reused to check the guard entailment precondition to the firing of each CHR rule.

Adding and altering CHR allows one to rapidly customize, extend, mix and match various solving methods. There are two main options for the CHR engine: it can interpret the CHR, or it can compile them into imperative style Prolog rules [Holzbaur and Frühwirth, 98]. These Prolog rules (together with the application CLP rules and the Prolog rules that define the CHR built-ins) are then in compiled to execution platform native code, through an intermediate level of CLP virtual machine code. The Prolog + CHR approach share all the adaptation, explanation and integration weaknesses of the Prolog + Library approach.

The Prolog + CHR architecture

- represents the solving task declaratively by CLP Horn rules;
- represents the solving method declaratively by CHR conditional rewrite rules and Prolog Horn rules;
- deploys the code as a hybrid CLP, CHR, Prolog rule base processed by a CLP-CHR engine which is accessible from an application via programming language bridges;
- uses Prolog rules to declaratively compile the CLP, CHR and Prolog rules into a CLP virtual machine code and then from such intermediate code to native code.

4.4 Prolog CHR$^\vee$

CHR$^\vee$ [Abdennadher S., 01] extends CHR with disjunctive bodies, *i.e.*, allowing rules of the form

$$S_1 ,..., S_a <=> G_1 ,.., G_b \mid (B_1^1,..., B_c^1) ;...; (B_1^g ,..., B_h^g) \text{ and}$$
$$P_1 ,..., P_d ==> G_1 ,.., G_e \mid (B_1^1 ,..., B_i^1) ;...; (B_1^j ,..., B_k^j)$$

with the expected corresponding logical semantics

$$G_1 \wedge ... \wedge G_b \Rightarrow (S_1 \wedge ... \wedge S_a \Leftrightarrow (B_1^1 \wedge ... \wedge B_c^1) \vee ... \vee (B_1^g \wedge ... \wedge B_h^g) \text{ and}$$
$$G_1 \wedge ... \wedge G_e \Rightarrow (P_1 \wedge ... \wedge P_d \Rightarrow (B_1^1 \wedge ... \wedge B_i^1) \vee ... \vee (B_1^j \wedge ... \wedge B_k^j).$$

Operationally, disjunctive bodies introduce the need for backtracking search in the CHR engine. When a disjunctive rule R triggers, one of its alternative bodies (B_l^m ,..., B_l^n) is chosen to be added to the constraint store and the engine then continues CHR forward chaining. However, if at a latter point, the store simplifies to `false`, instead of terminating, the CHR$^\vee$ engine then backtracks to (B_l^m,...,B_l^n) and deletes it from store together with all the constraints that were subsequently added (directly or indirectly) based on the its presence in the store. It then adds to the store the next alternative body (B_o^p,...,B_o^q) in R and resumes rule forward chaining. CHR$^\vee$ not only extends CHR with disjunctive bodies, but it also extends Prolog with multiple heads [Abdennadher S., 01]. Recall from Section 4.2 that the semantics of Prolog rules that share the same head is:

$$H \Leftrightarrow (G_1^1 \wedge ... \wedge G_r^1) \vee ... \vee (G_1^s \wedge ... \wedge G_t^s).$$

This precisely matches the semantics of the single head, CHR$^\vee$ rule: H <=> (G_1^1,..., G_r^1) ;...; (G_1^s ,..., G_t^s). Thus, CHR$^\vee$ is a single language that is more expressive to model constraint *tasks* than CLP and more expressive to model constraint *methods* than CHR. Using CHR$^\vee$ instead of CLP to model constraint tasks provides the power of full first order logic to define complex constraints from a minimum set of very primitive ones like `true`, `false` and = (syntactic equality between first order logic terms). Any first order logic formula F can be converted to a semantically equivalent formula N in the implicative normal form $P_1 \wedge ... \wedge P_n \Rightarrow B_1 \vee ... \vee B_o$ which is precisely the semantics of a guardless CHR$^\vee$ propagation rule.

All currently available implementations of CHR$^\vee$ are Prolog-based. Their architecture is shown in Figure 5. Similarly to the hybrid Prolog + CHR architecture, some implementations interpret the CHR$^\vee$ rules, while others compile them into imperative style Prolog rules. While this architecture uses CHR$^\vee$ instead of CLP rules to model the solving task and to implement the arbitrary symbolic constraints, it nevertheless still relies on Prolog rules to define the CHR$^\vee$ built-in constraints and on Prolog's naive CBT search to process disjunctive bodies and underconstrained finite domains. This makes them significantly slower than procedural libraries that rely on more efficient methods for these tasks such as **Conflict-Directed Backjumping** (CDBJ). This also makes them inherit the other already mentioned weaknesses of the Prolog-based CPP architectures.

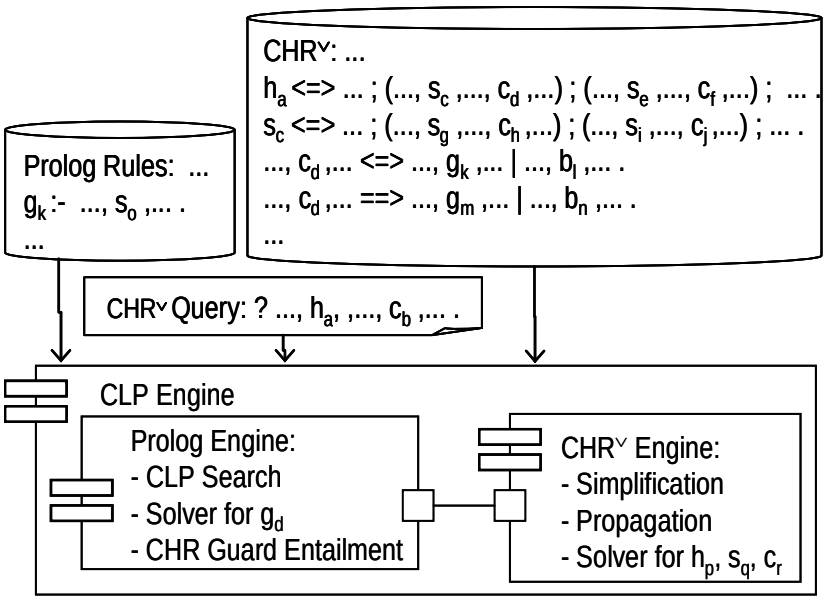

Figure 4: The Prolog with CHR CPP Architecture.

The Prolog CHR$^{\vee}$ architecture

- represents the solving task declaratively by CHR$^{\vee}$ disjunctive conditional rewrite rules;
- represents the solving method declaratively by CHR$^{\vee}$ disjunctive conditional rewrite rules and Prolog Horn rules;
- deploys the code as a hybrid CHR$^{\vee}$, Prolog rule base processed by a CLP-CHR engine which is accessible from an application via programming language bridges;
- uses Prolog rules to declaratively compile the CHR$^{\vee}$ and Prolog rules into a CLP virtual machine code and then from such intermediate code to native code.

Prolog engines SICStus [SICStus, 07] and ECLiPse [ECLiPSe CPS, 07] offer all three Prolog-based CPP architectures.

4.5 Compiling CHR to COIP API

This most recent CPP architecture, shown in Figure 6, attempts to combine the respective strengths of the COIP API and Prolog + CHR architectures by

- representing the solving task declaratively by CHR conditional rewrite rules;
- representing the solving method in part declaratively by CHR conditional rewrite rules and in part procedurally by COIP class operations;
- deploying the code as COIP objects.

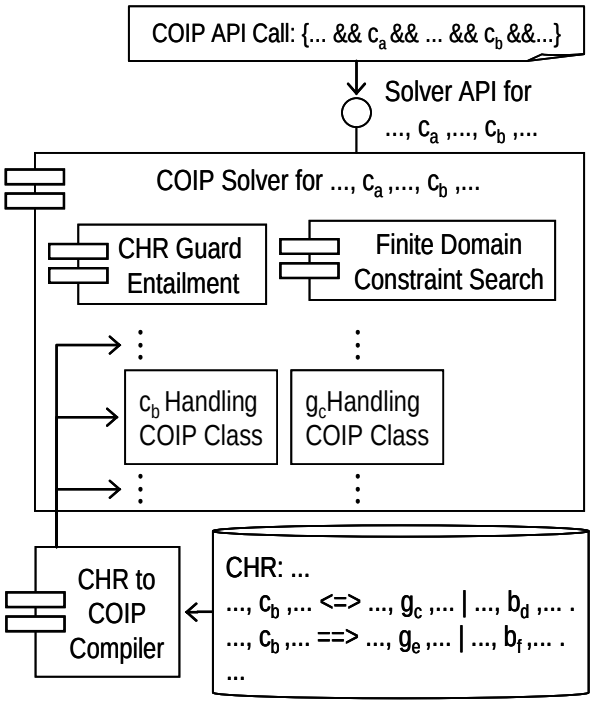

Figure 5: The CHR to COIP Compiling CPP Architecture.

These deployed objects result from a two stage compilation scheme: (a) compiling the CHR to COIP source code, followed by (b) compiling the resulting source code, together with the COIP classes that implement the CHR built-in constraint handlers, into deployable code. The compiling method is declaratively structured as COIP classes but its details are procedurally represented as operations. In this approach, a COIP such as Java substitutes Prolog as the underlying host language for the CHR [KULeuven, 07], [Wolf, 06]. This requires re-implementing from scratch in the COIP services that were provided by reusing Prolog built-ins: unification to check the CHR guard entailment condition and CBT for finite domain search. While requiring more work, this also creates new opportunities such as relying on more efficient specialized algorithms for these tasks *e.g.*, CDBJ instead of CBT. It also allows incorporating solution adaptation techniques such as **Justification-Based Truth-Maintenance** (JTMS) [Wolf, 00], where each constraint in the store is kept with the indexes of the rule which firing inserted the constraint in the store and of the justification for such firing, namely the unique identifiers of the RDC that matched the rule heads and the matching equations and BIC that entailed the rule guards. For example, firing the rule: $i@\ c(X,Y)_j \wedge d(X)_k \Longrightarrow (X = a)_m \mid e(Y)_n$, where i is the rule name and j, k, m and n are the respective unique identifiers of the rule's constraints, results in adding to the RDC the constraint $e(Y)_n$ with the justification {i, j, k, m}. The Java CHR engines JCHR [KULeuven, 07] and DJCHR [Wolf, 06]

follow this architecture, with the latter incorporating a JTMS scheme to provide efficient solution adaptation. As interface, they only provide a Java API. The lack of an interactive GUI makes them unpractical for rapid testing and application-specific customization and combination of solving methods.

5 A New Architecture for Constraint Programming Platforms

In the previous section, we have seen that all CPP proposed so far only very partially fulfill the six key requirements for a practical CPP. In this section, we propose a new CPP architecture that aims to simultaneously fulfill them all. The first key idea is to integrate in synergy three cutting-edge reuse fostering software architecture concepts: Component-Based Architecture, Aspect-Oriented Architecture and Model-Driven Architecture. In what follows, we first briefly introduce each of these concepts, before explaining how they can be combined and put into practice using currently available CASE tools. We then list seven principles to apply the combination of these three general software architecture concepts to our specific concern of conceiving a software architecture for a practical CPP that satisfies the six requirements for such a platform. These principles explain how to integrate these three software architecture concepts together with the most powerful CP concepts that we identified in section 4.

5.1 Cutting-Edge Reuse Fostering Software Architecture Concepts

To achieve low-cost portability to multiple execution platforms and automate more development process sub-tasks, a **Model-Driven Architecture** (MDA) [Stahl and Völter, 06] switches the software engineering focus away from low-level source code towards high-level models, metamodels (*i.e.,* models of modeling languages) and model transformations that automatically map one kind of model into another. It prescribes the construction of a fully refined **Platform Independent Model (PIM)** together with two sets of model transformation rules to translate the PIM into source code via an intermediate **Platform Specific Model (PSM)**.

To achieve low-cost evolution, a **Component-Based** MDA [Atkinson et al, 02] structures the PIM, PSM and source code as assemblies of reusable components, each one clearly separating the services interfaces that it provides to and requires from other components from its encapsulated realization of these services (itself potentially a recursive sub-assembly).

To achieve separation and reuse of **cross-cutting concerns** (*i.e.,* bits of processing that cannot be satisfactorily encapsulated in a single component following any possible assembly decomposition), **Aspect-Oriented** MDA [Stahl and Völter, 06] prescribes to model such concerns as PIM to PIM model transformations that weave the corresponding additional model bits at appropriate locations scattered throughout the main concern PIM.

Today, a component-based MDA can be fully specified using the UML2 standard [Eriksson, Penker et al, 04] for it incorporates (a) a platform independent component metamodel, (b) the high level object-oriented functional constraint language OCL2 [Warmer and Kleppe, 03] to fully detail, constraint and query UML2 models, and (c) the **Profile** mechanism to define, in UML2 itself, UML2 extensions with platform specific constructs for diverse PSM. Model transformations for PIM to PIM aspect

weaving and PIM to PSM to code translation can be specified using the rule-based, hybrid declarative-procedural Atlas Transformation Language (ATL) [ATL, 06], which reuses OCL2 in the left-hand and right-hand sides of object-oriented model rewrite rules. These rules are applied by the ATL Development Tool (ATLDT), conveniently deployed as an Eclipse Plug-in [Eclipse, 07]. In a **Component-Based Aspect-Oriented Model-Driven Architecture** (CBAOMDA), only the core concern PIM and the model transformations are hand-coded. The other models and the code are automatically generated from the core concern PIM by applying the transformation pipeline to it.

5.2 Software Architecture Principles for a Practical CPP

The new CPP architecture that we propose is based on seven principles to instantiate the general idea of CBAOMDA so as to integrate it with the CP concepts that our review of the field revealed as the most promising to fulfill the six key requirements of a practical CPP. These principles are the following:

1. To combine very expressive solving task modeling with rapid method customization and combination, use **CHR$^{\vee}$** as an uniform language to **declaratively represent both the solving task and the solving method** (except for a minimal set of CHR built-in constraint solvers represented by Boolean operations of COIP classes).
2. To combine solution adaptation with solution explanation generation, incorporate the **JTMS** techniques of DJCHR and extend them to deal with **disjunctive CHR**.
3. To provide rapid prototyping deploy the CHR$^{\vee}$ engine as an **Eclipse plug-in** [Eclipse, 07] with a GUI to **interactively submit queries and inspect solution explanations** at various levels of details.
4. To combine efficient solving implementation techniques with seamless integration in applications, build the first **CHR$^{\vee}$ to COIP compiler**.
5. To go one extra step towards reuse and extensibility, follow a an object-oriented, **component-based, model-driven architecture** for the overall CHR$^{\vee}$ engine.
6. To go one extra step towards easy to customize declarative code, define the CHR$^{\vee}$ compiler as a base of **object-oriented model transformation rewrite rules**.
7. To go one extra step towards separation of concerns, represent the solving **explanation generation** processing models **as orthogonal aspects** separate from the core solving model and incorporated them by using weaving model transformations.

5.3 An Adaptive CHR$^{\vee}$ to COIP Compiling CBAOMDA for CPP

The high-level blueprint of the new constraint platform architecture that we propose is given in Figure 7 where the hand-coded elements are highlighted in gray while those automatically generated through model transformation rules are in white.

As an MDA, it represents the system at four layers of abstractions: the PIM, the PSM, the source code and the deployed code. This last layer is omitted from Figure 7 for conciseness reasons. Each layer is entirely generated from the highest layer in a fully automated way. First the PSM is fully generated from the PIM by applying ATL model transformation rules using ATL-DT. Then, the source code is fully generated from the PSM by ATL code generation transformation meta-code. Finally, the deployed code is fully generated from the source code by the target COIP compiler and deployment IDE. Our architecture thus belongs to the most advanced, fully generative, model transformation based MDA. But even as compared to such advanced MDA, it further innovates in three ways:

- by dividing the PIM layer into a *declarative* rule-based sub-layer in CHR$^{\vee}$ and a *procedural* object-oriented sub-layer based on UML2 component and class diagrams, composite structure diagrams and activity diagrams, all three fully annotated by OCL2 constraints;
- by fully automatically generating part of the procedural PIM sub-layer from the declarative PIM sub-layer, more specifically by generating the procedural UML2/OCL2 constraint handler component model as output of a CHR$^{\vee}$ compiler;
- by implementing this compilation process itself as the application by the ATL-DT of a pipeline of three ATL model transformation rule bases.

The first element of this pipeline transforms the original CHR$^{\vee}$ base into a semantically equivalent base in a restricted core of CHR from which it is simpler to generate a semantically equivalent procedural representation. This first element includes transformations to axiomatize the disjunctions in the body of a CHR$^{\vee}$ rule into a CHR constraint (*e.g.*, by transforming a body `c1; ... ;cn` into a constraint `or(c1,...,cn)` where `or` becomes a constraint symbol and the constraint symbols of `c1,...,cn` become functions symbols. This allows taking the search necessary to deal with disjunctive bodies out of the constraint handler and into a separate search component. This separation of orthogonal concerns prevents the resulting solver to be limited to a single search strategy hard-wired in the constraint handlers. Instead, various search strategies can be assembled with the same handlers. The first element of the compiler pipelines also includes full CHR to core CHR transformations to simplify subsequent transformation of this core into a procedural handler model. One such transformation is to shift the constants in the heads as new guard equations to allow for a simpler, uniform procedural rule head matching process (*e.g.*, transforming the CHR `c(a,Y) <=> true` into the operationally equivalent CHR `c(X,Y) <=> X = a | true`).

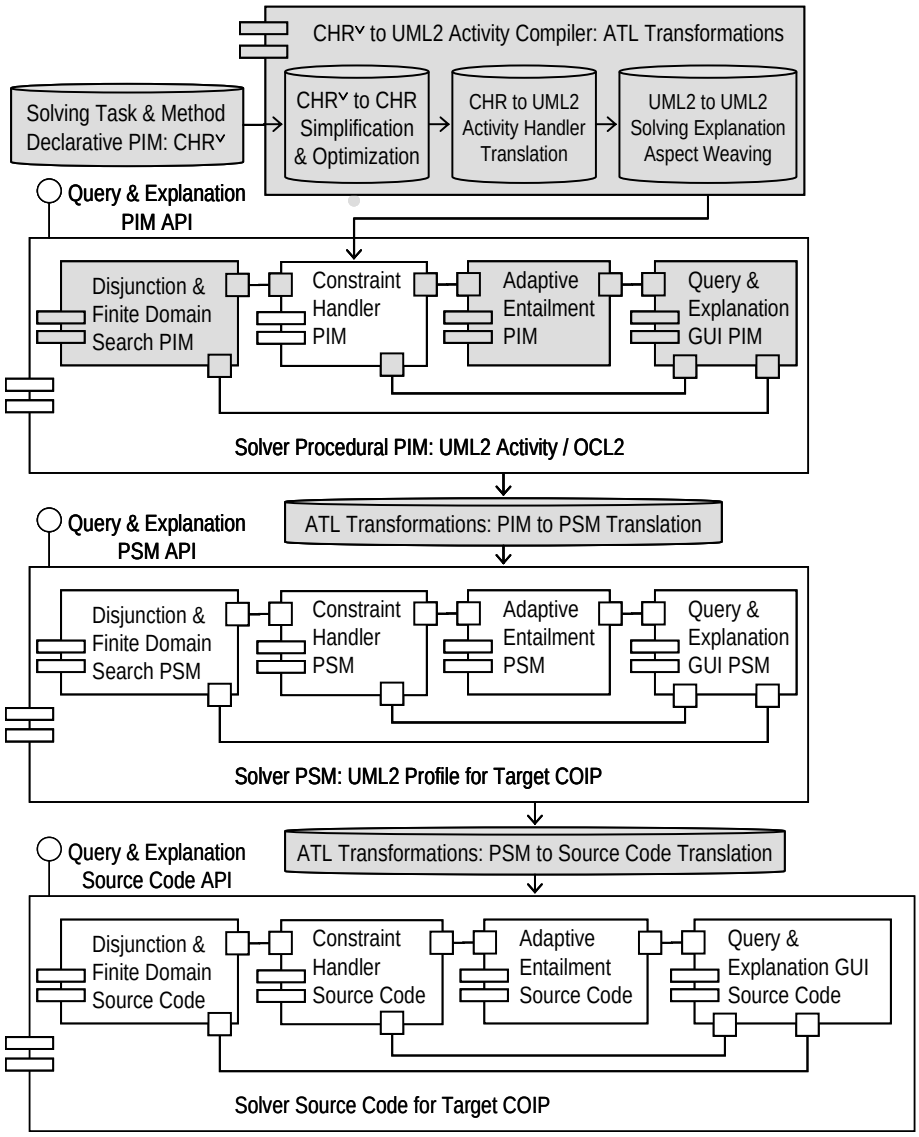

Figure 6: Our New CHR∨ to COIP Compiling CBAOMDA for CPP.

The second element of the pipeline in the CHR$^\vee$ to UML2 compiler transforms the core CHR into a procedural UML2 model. This model however concerns itself exclusively with efficiently solving constraints in an adaptive manner by reusing the JTMS techniques of DJCHR. It ignores the orthogonal concern of generating a trace of this adaptive solving process to be used for reasoning explanation purposes. This orthogonal concern is represented as a *PIM level aspect*. It is inserted onto the UML2 procedural constraint handler PIM by the third element of the pipeline that consists of transformations that weave the UML2 elements modeling the reasoning trace generation activity into specific point cuts scattered among the UML2 elements modeling the constraint handling activity. It makes our architecture *aspect-oriented* in addition to model driven. Finally, our architecture is also *component-based*: at the four procedural layers (PIM, PSM, source code and deployed code), it consists of an assembly of four components:

- the constraint handler that encapsulates the constraint simplification and propagation techniques;
- the search component needed to process CHR$^\vee$ disjunctions and finite domains not reducible to singletons by the handler;
- the adaptive entailment component to determine which CHR$^\vee$ can be fired given the current state of the constraint store;
- a GUI to type in queries with which to initialize the store and provide explanation for the solver answer and reasoning.

This innovative architecture reconciles:

- expressive solving task modeling in CHR$^\vee$;
- rapid method customization and combination in CHR$^\vee$;
- efficient method implementation by compiling CHR$^\vee$ onto procedural COIP classes which are then compiled to bytecode or native code;
- solution adaptation by incorporating JTMS;
- solution explanation by generating a solving trace exploiting the JTMS justifications and featuring a trace browsing GUI;
- seamless integration in application by providing the solving services as a COIP interface.

It thus promises to be the first to fulfill all six key requirements of a CPP.

Table 1 sums up how each CPP architecture fulfils these requirements.

	COIP API	Prolog + Procedural Handler Library	Prolog + CHR	Prolog CHR$^{\vee}$	CHR to COIP Compiler	CBAOMD CHR$^{\vee}$ to COIP Compiler
Task Expression	*	**	**	***	**	***
Method Customization	*	*	**	**	**	***
Efficient Implement.	***	**	*	*	*	**
Solution Adaptation	*	*	*	*	***	***
Solution Explanation	*	*	**	**	*	***
Seamless Integration	**	*	*	*	**	***

Table 1: Compared fulfillment of CPP requirements by CPP architectures.

6 Conclusion and Future Work

In this paper, we identified six key requirements for a versatile and practical CPP. We used these requirements to critically review five prominent architectures among currently available CPP. We showed that each of them fails to adequately fulfill roughly half of these requirements. There is thus much room for improvement in the field of CPP architecture. To contribute to such improvement we proposed an innovative CPP architecture that

- integrates in synergy rule-based, component-based, object-oriented, aspect-oriented and model-driven architectural principles;
- compiles CHR$^{\vee}$ rules into COIP source code, in several stages, using UML2 as an intermediate language;
- incorporates JTMS techniques to support efficient solution adaptation and explanation;
- includes a GUI to pass as query input a constraint task instance and browse the generated justification-based explanation at various levels of detail;
- also includes an API to seamlessly provide the same query, solving and explanation facility to external software.

We discussed why such architecture is the first one ever proposed with the potential to fulfill all key six requirements of a CPP. Confirming such potential will require implementing a running prototype of the architecture, measuring its efficiency against state of the art solvers on benchmark tasks and integrating it in within various practical applications for usability validation. We now stand midway towards the first

of these goals. We have developed in UML2/OCL2 the fully refined PIM of an adaptive guard entailment component, as well as the ATL rules to compile a CHR^{v} base into a constraint handler PIM in UML2/OCL2. The detail of this compiler is the object of another publication currently in preparation. The pieces still missing from puzzle are the PIM for the search and GUI components, as well as the ATL rules for the PIM to PSM and PSM to source code translations. Together, our innovative architecture and its implementation will make practical contributions to several fields beyond constraint programming. To automated reasoning, it will provide the first scalable yet highly versatile base component on top of which to assemble and integrate deduction, abduction, default reasoning, inheritance, belief revision, belief update, planning and optimization services. To innovative programming language compiler and run-time system engineering, it will show the benefits of the component-based, aspect-oriented and model-driven approaches. It will also extend the scope of application of these approaches by showing that their practical benefits are even greater for cutting edge systems that perform intelligent processing with high aggregated value than for the straightforward GUI to database back to GUI translations performed by the standard web information systems for which these approaches were initially conceived.

References

[Abdennadher S., 01] Abdennadher S. Rule-based Constraint Programming: Theory and Practice. Habilitationsschrift, Institut für Informatik, Ludwig-Maximilians-Universität München, 2001.

[Ait-Kaci and Nasr, 86] Ait-Kaci, H., Nasr, R. LOGIN: A Logic Programming Language with Built-in Inheritance. In Journal of Logic Programming, 3:185--215. 1986.

[Atkinson et al, 02] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J. and Zettel, J. Component-based Product Line Engineering with UML. Addison-Wesley, 2002.

[ATL, 06] The ATL User Manual. http://www.eclipse.org/gmt/atl/doc/.

[CHIP, 07] COSYTEC CHIP V5.8, http://www.cosytec.fr/news/chip_v5.8.htm.

[Costa et al, 03] Costa, V.S., Page, D., Qazi, M., Cussens, J. CLP(BN): Constraint Logic Programming for Probabilistic Knowledge. In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence. Acapulco, Mexico. 2003.

[Eclipse, 07] Eclipse: An Open Development Platform. 2007, http://www.eclipse.org/

[ECLiPSe CPS, 07] The ECLiPSe Constraint Programming System. 2007, http://eclipse.crosscoreop.com/.

[Eriksson, Penker et al, 04] Eriksson, H.E., Penker, M., Lyons, B., Fado, D. UML 2 Toolkit. Wiley. 2004.

[Holzbaur and Frühwirth, 98] Holzbaur, C., Frühwirth, T. Compiling Constraint Handling Rules. In Proceedings of the ERCIM/COMPULOG Workshop on Constraints, Amsterdam. 1998.

[Frühwirth, 94] Frühwirth, T. Theory and Practice of Constraint Handling Rules. In Journal of Logic Progamming. 19(20), 1994.

[Jim and Thielscher, 06] Jin, Y., Thielscher, M. Iterated Belief Revision, Revised. In Artificial Intelligence. (to appear), 2006.

[KULeuven, 07] KULeuven JCHR. 2007. http://www.cs.kuleuven.be/~petervw/JCHR/.

[Marriott and Stuckey, 98] Marriott, K. and Stuckey, P. Programming with Constraint: An Introduction. MIT Press. 1998.

[Puget, 94] Puget, J.F. A C++ Implementation of CLP. In Proceedings of the Second Singapore International Conference on Intelligent Systems. Singapore. 1994.

[Robin and Vitorino, 06] Robin J., Vitorino, J. ORCAS: Towards a CHR-Based Model-Driven Framework of Reusable Reasoning Components. In Proceedings of the 20th Workshop on (Constraint) Logic Programming (WLP'06). Vienna, Austria, 2006.

[Russell and Norvig, 03] Russell, S. and Norvig, P. Artificial Intelligence: A Modern Approach (2nd Ed.). Prentice-Hall. 2003.

[SICStus, 07] SICStus Prolog. 2007, http://www.sics.se/isl/sicstuswww/site/index.html

[SWI, 07] SWI Prolog, www.swi-prolog.org/

[Stahl and Völter, 06] Stahl, T., Völter, M. Model-Driven Software Development: Technology, Engineering, Management. Wiley. 2006.

[Thielscher, 02] Thielscher, M. Programming of Reasoning and Planning Agent with FLUX. In Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), Toulouse, France. 2002.

[Warmer and Kleppe, 03] Warmer J., Kleppe A. The Object Constraint Language (2nd Ed.): Getting Your Models Ready for MDA, Addison-Wesley, 2003.

[WebCHR, 07] 2007. WebCHR: http://www.cs.kuleuven.be/~dtai/projects/CHR/.

[Wolf, 06] Wolf, A. Object-Oriented Constraint Programming in Java Using the Library firstcs. In Proceedings of the 20th Workshop on (Constraint) Logic Programming (WLP'06).Vienna, Austria, 2006.

[Wolf, 01] Wolf, A. Adaptive Constraint Handling with CHR in Java. In Lecture Notes in Computer Science, 2239, Springer, 2001.

[Wolf, 00] Wolf, A., Gruenhagen, T., Geske, U. On Incremental Adaptation of CHR Derivations. In Journal of Applied Artificial Intelligence 14(4). Special Issue on Constraint Handling Rules. 2000.

[XSB, 08] XSB Prolog. http://xsb.sourceforge.net/

[YAP, 07]. Yet Another Prolog. www.ncc.up.pt/~vsc/Yap