Posting and Retracting Constraints in Cooperating Asynchronous Solvers

Georg Ringwelski

Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany
georg.ringwelski@first.fhg.de

Abstract. We define a new incremental execution model for constraint satisfaction in concurrent imperative programming. Cooperating solvers restrict variable domains and detect inconsistecies by executing propagation and search algorithms asynchronously to the application program in which they were posted. Non-deterministic labeling-constraints, which implement arbitrary search algorithms, or interactive applications use constraint retraction to withdraw erroneous instantiations or to solve dynamic CSPs. The denotational semantics of such constraint programs is defined as the result of the Chaotic Iteration of domain reduction functions that are implied by the propagation methods of posted (and not retracted) constraints.

1 Introduction

Since it was introduced in the early 1980s Constraint Programming (CP for short) has reached many application areas of Artificial Intelligence. Providing several algorithms for the efficient solving of Constraint Satisfaction Problems (CSP), it has become an important point of interest for the AI community. The success of Constraint Programming is based largely on the efficiency of the algorithms used. Restricting variable domains with propagation followed by search in a thus reduced search space would seem to be a very powerful concept for finding solutions to combinatorial problems. In order to establish these techniques in industrial software systems, it is about time to integrate constraints into a widespread object-oriented programming language for the use of programmers that are no CP experts [9]. Besides ILOG's [13] commercial tools for CP in C++ (ILOG Solver) or Java (ILOG JSolver), some academic Java libraries for constraint solving exist [8, 10, 18, 12]. Additionally, most CLP implementations (e.g. SICStus [3]) provide interfaces to popular object-oriented programming languages.

Our idea is to use the efficient problem-solving mechanisms from CSP, i.e. algorithms for propagation and search, in a generic programming library that is well suited for use in object-oriented interactive software systems. We define an incremental execution model for constraint processing in imperative programs that allows a user resp. an application program to interact asynchronously with the process of constraint solving. The constraint solvers (possibly running in parallel) are used as black-boxes, while interactive applications post and retract constraints asynchronously in the solver and read inferred information from variable domains. To allow users to withdraw former input and to escape from erroneous branches in the search tree of NP-complete problems we use constraint retraction. By retracting constraints alternative states on other branches in the search tree are computed from locally available information. We hope, that an implementation of our concept will allow the integration of CP as a problem solving component into complex interactive software systems, enabling constraint technology to spread more widely.

1.1 Constraint Programming Environments

In a black-box constraint solver the internal inference should be hidden from the programmer who uses the component. The output of such a solver can be read in variable-objects, while the input is made by an operation that adds new constraints to the knowledge base and another operation that removes previously added constraints. We think, that any user interaction concerning the constraints in interactive applications, can be interpreted by these two operations which allow to incrementally adapt the CSP. Because every interaction is propagated immediately, the user can see the consequences of her decisions shortly after her interaction and can thus be guided by the system on her search for the best solution in the often unmanageable set of solutions for her initial problem. The application program can be used to interactively figure out a sufficient (wrt. correctness and details) constraint-based model of the user's real world problem. Providing a usable model of their problem is, according to our experience, a very hard task for many users. Non-incremental solvers like ILOG Solver 5.0 [13], JCL [8] or Local Search solvers do not support such an interactive definition and incremental solving of dynamic CSPs.

In constraint solvers that use state restoration techniques (e.g. trailing) constraint retraction can be implemented by restoring a former state from some stored history and reposting all constraints that were posted since then. Upward trailing [4], which finds former states by upward state restoration (backtracking) in the search tree, is used for the necessary history management. This techniques is used in most CLP solvers like SICStus [3], clp(R) [7] or ECL^{i}PS^e [17], and also in most OOP solvers like JaCK [10] or ILOG [13]. In order not to use a central history management like in upward trailing, alternative downward directed restoration techniques, applicable in distributed CP, are investigated for Mozart [5, 16] or Figaro [11, 4]. However, these new techniques stick to the idea that former states are rebuilt and the action that was performed since then is re-performed to remove erroneous input from the knowledge base.

We provide an operation for incremental constraint retraction, like that available in DJCHR [18] or CLP(fd,S) [6], to reverse erroneous input. In difference to the mentioned state-restoring systems we do not compute (neither upward nor downward directed) "former" states, but states "that would be the current state, if something were *not* done in the past". Unlike CLP(fd,S) or DJCHR we do not use a centralized history trail or constraint store for this task. This allows constraint retraction in concurrent applications where multiple solvers cooperatively solve a Distributed Constraint Satisfaction Problem (DCSP).

The incremental constraint solver Mozart [5] can also solve DCSPs. However, propagation has to be reduced to keep the solver's performance in distributed applications acceptable because it is performed synchronously with every domain restriction. The slow inter-process communication is limited in Mozart by performing propagation across process borders only for forward-checking but not for look-ahead propagation. Such restrictions to consistency algorithms are not necessary in our execution model. Since the execution of constraints is performed asynchronously, all communication between variables and constraints (i.e. repostings during propagation) can be performed with non-blocking message passing. A solver running in process A can continue his work without knowing that a solver in some other process B has finished a task that has been addressed from A to B.

Another important property of a constraint solver that is to be used as a knowledge management and inference component in complex software systems is the extendibility with new constraints and the compatability with new constraint systems. We decided to allow the user to implement new constraints for special applications that can be used for constraint satisfaction in heterogeneous problems. Similar to most established solvers, like ECL^{i}PS^e[17] for example, or in CHR-systems, like JaCK [10], we provide a framework for the integration of new constraints.

1.2 Preliminaries

Firstly we introduce some useful notation. ε is a mapping from vectors to sets, such that all components of the argument vector are elements of the result: $\varepsilon(\overline{x}) = \varepsilon(\langle x_1, ..., x_n\rangle) := \{x_1, ..., x_n\}$. To represent the changes of variable values over time we denote with v^t the value of v at time point t and with v^{t+1} its value at the time point after some described evaluations have been executed.

A constraint variable v is associated to a domain by $\mathsf{dom}(v) \subseteq \mathsf{dom}^*(v)$, where $\mathsf{dom}(v)$ is called the current domain of v and $\mathsf{dom}^*(v)$ the initial domain of v. Given the constraint variables $v_1, ..., v_n$ with $\mathsf{dom}(v_i) \subseteq \mathsf{dom}^*(v_i) =: D_i$, a constraint c over $\mathsf{vars}(c) = \langle v_1, ..., v_n\rangle$ is a set of vectors $\mathsf{sem}(c) \subseteq D_1 \times ... \times D_n$ representing valid variable bindings, $\mathsf{sem}(c)$ is called the declarative semantics of c. A sequence of constraints C induces the scheme $\mathsf{scheme}(C)$ that contains each variable occurring in any $c \in \varepsilon(C)$ exactly once. $\mathsf{scheme}(C)$ is uniquely determined by the order of C and $\mathsf{vars}(c)$ for each $c \in C$. A CSP is given by a sequence of constraints (that induce a scheme). The objective of constraint propagation for a CSP C with scheme $\langle v_1, ..., v_n\rangle$ is to reduce the (current) variable domains $\mathsf{dom}(v_i)$ as far as can be deduced from specific knowledge on $\mathsf{dom}^*(v_i)$. The CSP is called consistent, iff $\forall v_i \in \varepsilon(\mathsf{scheme}(C)) : \mathsf{dom}(v_i) \neq \emptyset$ and $\forall c \in \varepsilon(C) : \langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n)\rangle \downarrow \mathsf{vars}(c) \in \mathsf{sem}(c)$ hold; otherwise C is called inconsistent. To obtain a solution of a CSP C we extend C with an instantiation constraint i_v for each variable $v \in \bigcup_{c\in C} \varepsilon(\mathsf{vars}(c))$ and get the CSP $C' = C \circ \langle i_{v_1}, ..., i_{v_n}\rangle$. If C' is consistent, the vector $\langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n)\rangle$ is a solution of C.

2 Asynchronous Constraint Solving

Asynchronous Constraint Solving (ACS for short) is an execution model that allows the use of multiple solvers that cooperatively solve CSPs by constraint propagation. ACS does not make use of any system inherent search but requires encapsuled algorithms for (non-deterministic) search that post and retract instantiation constraints until a solution is found. The main difference to other CP execution models is that the execution of constraint posting and retraction is performed asynchronously from the tasks that implement these methods. The solvers buffer these tasks and execute them in concurrently running processes or threads.

2.1 Components for ACS

Constraint variables are associated to a representation of their domains. This representation is given as a vector of features (properties) that defines a set, i.e. the current domain of the variable. For example, a finite interval of integers can be represented by its minimum and maximum value leading to the representation $(min : int, max : int)$. If such a feature is changed, constraints depending on it should be reposted (propagation) in order to approximate consistency in the constraint network that is defined by a CSP.

Definition 1 (constraint variable). *A (constraint-) variable* $v = (\overline{p}, C)$ *of an initial domain* $D = \mathsf{dom}^*(v)$ *is given by*

1. *a vector* $\overline{p} \in P = P_1 \times P_2 \times ... \times P_n$ *of properties of* D*, such that a mapping* $\mathsf{dom} : P \to \mathcal{P}ow(D)$ *exists.*
2. *a family* $C = (C_p)_{p \in \{P_1,\ldots,P_n\}}$ *of sequences of constraints* C_p

For easier notation dom *is also defined for variables:* $\mathsf{dom}((\overline{p}, C)) = \mathsf{dom}(\overline{p})$.

Every constraint holds references to its variables and the solver it was posted in. The declarative semantics of every constraint is implemented in weighted propagation tasks that restrict variable domains and check consistency. In Fig. 2 for example we show the propagation task of the less-than constraint over finite integer domain variables which implements the declarative semantics of $<$ by reducing the domain of its variables or by creating an inconsistecy, if there are no appropriate values left in their domains. Furthermore, the consequences and the history of the constraint are stored, providing necessary information for constraint retraction. The consequences of a constraint c are constraints that were reposted within the propagation of c. These consequences must be reconsidered, if c is retracted. The history is a snapshot of the current domains of the constraint's variables at its first execution. The history information is used to compute domain extensions during constraint retraction.

Definition 2 (ACS constraint). *An ACS constraint* $c = (\overline{v}, T, acs, s, E, \overline{h})$ *is given by*

1. *a vector* $\overline{v} = \langle v_1, ..., v_n \rangle$ *of its variables*
2. *a set* $T = \{(t_1, p_1), ..., (t_m, p_m)\}$ *of propagation tasks where* t_i *are constraint tasks (see Def. 6) and* $p_i \in \{\bot, ..., \top\}$ *are their associated priority values, such that* $i \neq j \Rightarrow p_i \neq p_j$ *holds.*
3. *an asynchronous constraint solver acs*
4. *a current state* $s \in \{active, inactive\}$
5. *a set* E *of constraints called c's consequences*
6. *a vector* $\overline{h} = \langle \overline{p}_1, ..., \overline{p}_n \rangle$ *with* $\mathsf{dom}(\overline{p}_i) \subseteq \mathsf{dom}^*(v_i)$ *of domain descriptions called c's history*

For easier notation, the set of all propagation tasks $\{t_1, ..., t_m\}$ *is called* $\mathsf{tasks}(c)$ *and c's variables are denoted with* $\mathsf{vars}(c) = \varepsilon(\overline{v})$*.*

If a constraint fails, i.e. one of the constraint tasks detects an inconsistency, the system must provide a means to react to this inconsistency. The adequate reaction depends on the current computation context. For example, during search for discrete solutions an instantiation should be changed, during user interaction some output should be produced that asks the user for some decision, and so on. Therefore we define a general notion for such reacting components that transforms an inconsistent program state, i.e. an inconsistent CSP, to a consistent one by removing some constraints. The existence of such a transformation is guaranteed because at least the trivial removing of all constraints will lead to a consistent CSP, i.e. the empty sequence.

Definition 3 (repair-operation). *A repair-operation* ρ *maps an inconsistent CSP* C *to a consistent CSP* $\rho(C)$ *by removing constraints:* $\varepsilon(\rho(C)) \subset \varepsilon(C)$*.*

The execution of the propagation tasks given with every constraint is organized by a constraint solver that in our approach runs in parallel to the application. The parallel execution of the constraint solving process can be continued while the main application is busy with other tasks like user interaction for example. This allows to execute also time consuming propagation algorithms, if some spare time occurs because of (slow) user input. In ACS such expensive propagation algorithms can be integrated in low-priority propagation tasks, that are only executed, if no other evaluations are to be made. The solver always provides information on the progress of the current constraint propagation, which may be used by clients to decide when variable domains are read for further evaluations or output. This is expressed in the consistency-flag and the reliability value of the solver. The reliability is computed from the amount of un-executed solver tasks and their corresponding priorities. The reliability will reach $\top$ if no tasks are left to be evaluated; it decreases when constraints are posted or retracted and increases when tasks have been executed.

Definition 4 (Asynchronous Constraint Solver). *An asynchronous constraint solver* $s = (B, cons, \rho)$ *is given by*

1. *a task buffer* $B = \langle t_1, ..., t_n \rangle$ *where* $t_i = post(c, p) \vee t_i = retract(c)$
2. *a consistency flag* $cons \in \{consistent, inconsistent\}$
3. *a repair-operation* ρ*.*

The reliability value of s is evaluated from its buffer: $\mathsf{reliability}(B) \mapsto \{\bot, ..., \top\}$*.*

2.2 Constraint Programming with ACS

Given implementations, i.e. propagation tasks, of the required constraints, any CSP can be formulated in an imperative program by defining variables and posting constraints over them in running solvers. The CSP implied by a constraint program contains all posted constraints.

Definition 5 (constraint program). *A constraint program is an imperative program using the primitives 1-5 that are executed as follows:*

1. *solver declarations* $s :=$ `new Solver(`ρ`)` *create a new (consistent) solver* s *with empty task buffer that will call* ρ *if an inconsistecy is detected. The solver is started in a separate thread that executes the program shown in Fig.1:* $s = (\langle\rangle, consistent, \rho)$; s.`start()`
2. *variable definitions* $v :=$ `new Variable(`$\overline{p}$`)` *create new variables* v *with an initial domain description* $\overline{p}$ *and empty sets of depending constraints for all properties* P_i*:* $v = (\overline{p}, \{\emptyset_{P_1}, ..., \emptyset_{P_n}\})$
3. *constraint definitions* $c :=$ `new Constraint(`$\overline{v}, T$`)` *create a new constraint* c *that is implemented in the propagation tasks* T *over the variable vector* $\overline{v}$*. Initially* c *is not assigned to a solver, it is inactive and it has no consequences or history yet:* $c = (\overline{v}, T, null, inactive, \emptyset, null)$
4. *constraint postings* c.`post(`s`)` *activate a constraint* c *by adding its tasks* T *to the buffer of a solver* s*. Furthermore,* c *stores a snapshot of the domains of its variables to its history, i.e. it makes the following transformation:*
$c^t = (\langle v_1, ..., v_n\rangle, T, null, inactive, \emptyset, null) \wedge s^t = (B, cons, \rho) \longrightarrow$
$c^{t+1} = (\langle v_1, ..., v_n\rangle, T, s, active, \emptyset, \langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n)\rangle)$
$\wedge\, s^{t+1} = (B \dot\cup \bigcup_{(i,p)\in T} post(c,p), cons, \rho)$
5. *constraint retractions* c.`retract()` *deactivate a constraint* c *and push a corresponding retraction task to its solver's buffer, i.e. it makes the following transformation:* $c^t = (\overline{v}, T, s, active, E, \overline{h}) \wedge s^t = (B, cons, \rho) \longrightarrow$
$c^{t+1} = (\overline{v}, T, s, inactive, E, \overline{h}) \wedge s^{t+1} = (B \dot\cup \{retract(c)\}, cons, \rho)$

Where $B \dot\cup T$ *stands for the insertion of tasks* T *into a buffer* B*.*

Remark 1. We do not define any particular data structure for the task buffer of the solvers in Def. 5. The solver thread (Fig. 1, lines 7–14) always executes the first task in the buffer. The insertion of new tasks in the buffer (denoted by $\dot\cup$ in Def. 5.4 and 5.5) can be varied and is very relevant for the solver's performance.

Any desired constraint can be implemented in constraint tasks using domain-specific knowledge in algorithms that restrict variable domains or define, post or retract other constraints. The possibility to post or retract constraints within a constraint task is not supported in most CP systems because of termination problems. The possibility to post constraints as a side-effect of another constraint may lead to infinite chains of recursive constraint creations. We provide this possibility to have additional expressiveness which is (among others) very useful for the implementation of heterogeneous- or search-constraints. We expect

```
1  while run = true do                          // run until explicitely stopped by run:=false
2       if z = inconsistent                                     // an inconsistency occurred
3          then ρ.repair(); z := consistent;          // remove inconsistency somehow
4           else if (B = ∅)                                        // if nothing to do...
5                  then  wait until B ≠ ∅;                                   // ...wait
6                   else              // B = ⟨t1, t2, ..., tn⟩, i.e. tasks to execute
7                        B := ⟨t2, ..., tn⟩;                  // remove task from buffer
8                        if t1 = post((v̄, T, acs, s, E, h̄), p)    // a constraint posting
9                           then taski.execute() with (taski, p) ∈ T//run resp. task
10                       fi
11                       if t1 = retract(c)                   // a constraint retraction
12                          then c.retract()
13                       fi
14             fi
15      fi
16 end
```

Fig. 1. The loop of a concurrently running solver $acs = (B, z, \rho)$ that is started by acs.`start`(). If an inconsistency occurs it is removed by the solvers repair-operation. Solver tasks either run a constraint task (line 9), which is defined in Def. 6, or the constraint retraction (line 12), which is defined in Fig. 4.

the programmer not to implement infinite chains of constraint postings in her constraints.

The implementation of domain restrictions (Def. 6.1) in constraint tasks will approximate hyper-arc-consistency in the CSP that is implied by the respective program. For the most CSPs it is not practicable to evaluate hyper-arc-consistecy, such that the reliability values and the internal scheduling of the constraint tasks can be used to recognize the achievement of any other consistecy in the solver. We will describe this in Section 4. Inconsistencies are detected in constraint tasks (or in variables) whenever a current variable domain is reduced to the empty set.

Definition 6 (constraint task). *The tasks $t \in \mathsf{tasks}(c)$ of a constraint $c^t = (\overline{v}, T, acs, s, E, \overline{h})$ are imperative programs that use the primitives 1-5 which are executed follows.*

1. *domain restrictions $Prop := v.\mathtt{in}(\overline{p}')$ do not change the system, if the former domain is part of the newly defined, i.e. $\mathsf{dom}(\overline{p}') \supseteq \mathsf{dom}(v)$.*
   Otherwise v's domain is restricted to a representation of the intersection of old and new domain by changing the corresponding properties. In that case c is added to v's lists of constraints depending on property P_i where p_i has been changed due to the intersection, i.e. the domain restriction transforms every $v \in \overline{v}$ the following way:
   $v^t = (\langle p_1, ..., p_m\rangle, \{C_{P_1}, ..., C_{P_m}\}) \longrightarrow v^{t+1} = (\langle p''_1, ..., p''_m\rangle, \{C'_{P_1}, ..., C'_{P_m}\})$, *such that*
   $$(p''_i, C'_{P_i}) = \begin{cases} (p_i, C_{P_i}), & \textit{if } \mathsf{dom}(\langle p_1, ..., p_i, ..., p_m\rangle) \subseteq \mathsf{dom}(\langle p_1, ..., p'_i, ..., p_m\rangle), \\ (p'_i, C_{P_i} \cup \{c\}), & \textit{else} \end{cases}$$
   If v's domain is restricted wrt. property p_i, all constraints in C_{P_i} are reposted

in their solver and returned in the set $Prop$*, which is unified with* E *to make up the new set of consequences of* c*:* $c^{t+1} = (\overline{v}, T, acs, s, E \cup Prop, \overline{h})$

2. *inconsistency detections:* $acs.$`inconsistency`$()$ *set the consistency flag of the solver* acs *to inconsistent.*
3. *constraint definitions* $d :=$ `new Constraint`$(\overline{v}', T')$ *create a new constraint like in Def. 5.3 but define its solver to be* acs*:* $d = (\overline{v}', T', acs, inactive, \emptyset, null)$
4. *constraint postings* $d.$`post`(acs) *are executed as in Def. 5.4.*
5. *constraint retractions* $d.$`retract`$()$ *are executed as in Def. 5.5.*

Such constraint task implementations can be started by the call of $t.execute()$*, as shown in Fig. 1, line 9.*

To clarify the algorithm for domain restrictions (Def. 6.1) we show its imperative implementation in Fig. 3. Additionally we give a small example for the implementation of a constraint task in Fig. 2.

if $min_1 >= max_2$ — v_1 cannot be less than v_2
  then $mySolver.$`inconsistency`$()$
   else $E.addAll(v_1.\texttt{in}(\langle min_1, max_2 - 1\rangle));$ — restric domains and...
        $E.addAll(v_2.\texttt{in}(\langle min_1 + 1, max_2\rangle));$ — ...save propagated constraints in E
fi

Fig. 2. The example implementation of the only task of a less-than constraint $v_1 < v_2$ over finite integer domain variables $v_1 = (\langle min_1, max_1\rangle, \{C_{MIN_1}, C_{MAX_1}\})$ and $v_2 = (\langle min_2, max_2\rangle, \{C_{MIN_2}, C_{MAX_2}\})$.

Remark 2. Only constraints that were defined in a constraint task (Def. 6.3) can be posted or retracted (Def. 6.4 and 5) in the same task, because there is no other way to get a reference to a constraint here. Thus the possibility to define cyclic constraint dependencies does not arise.

For the formal discussion on the semantics for ACS in Sec. 3 we introduce some further notation. Some of the functions in Def. 7 can only be evaluated *after* program execution. In non-deterministic programs, such as most interactive applications, there is no way to define the set of all posted constraints in advance. These a posteriori evaluated sets will only be used to analyze the formal semantics of constraint programs and are not used operationally.

Definition 7 (properties of constraint programs and tasks). *The set of all solvers declared in a program* P *is called* $\mathsf{solvers}(P)$*. The set of all constraints that were posted during an execution of* P *is denoted by* $\mathsf{constr}(P)$*. The constraints that were defined (Def. 6.3) and posted (Def. 6.4) in a constraint* c *are subsumed in* $\mathsf{builtin}(c)$*. A constraint* c *is called* additive*, iff all* $c' \in \mathsf{builtin}(c)$ *are not retracted. A constraint program* P *is called* additive*, iff all* $c \in \mathsf{constr}(P)$ *are additive and not retracted in* P*. The set of all domain restrictions occurring in the tasks of a constraint* c *is called* $\mathsf{red}(c)$*. A constraint program is called consistent, if in its execution, no inconsistencies (Def. 6.2) were detected. The scheme of a constraint program* P *that posts the constraints* $c_1, ..., c_n$ *(in that order) is given by* $\mathsf{scheme}(P) := \mathsf{scheme}(\langle c_1, ..., c_n\rangle)$*.*

$Prop := new\ ConstraintSet();$ — to store reposted constraints
foreach $p'_i \in \varepsilon(\overline{p}')$ **do** — check every property of v's domain
  if $\mathsf{dom}(\langle p_1, ..., p'_i, ..., p_n\rangle) \subset \mathsf{dom}(\langle p_1, ..., p_i, ..., p_n\rangle)$
    then $p_i := p'_i$; — reduce domain
      foreach $c' = (\overline{v}, T, s, active, E, \overline{h}) \in C_{P_i}$ — dependent, active constraints
        $c'.post(acs)$; — repost, i.e. propagate
        $Prop := Prop \cup \{c'\}$ — store as consequence
      end
      if $c \notin C_{P_i}$ — if not already contained
        then $C_{P_i} := \langle C_{P_i}, c\rangle$ — insert c as depending constraint
      fi
  fi
end
return $Prop$; — return all consequences

Fig. 3. The algorithm to execute a call $Prop := v.\mathtt{in}(\langle p'_1, ..., p'_n\rangle)$ for domain restriction in a variable $v = ((p_1, ..., p_n), (C_{P_1}, ..., C_{P_n}))$.

2.3 Constraint Retraction

Recapitulating on the previous section, the call of $c.\mathtt{post}(acs)$ of a constraint c may cause three changes in the system:

(1) The domain of any $v \in \mathsf{vars}(c)$ may be reduced.
(2) Previously posted constraints with common variables may be reposted owing to the reduction of $\mathsf{dom}(v)$ (constraint propagation).
(3) The thus reduced variable domains may be used in later constraint executions.

If c is retracted, these changes must be reversed in the system. Our constraint retraction algorithm, shown in Figure 4, is a mixture of the algorithms for retraction in clp(fd,S) [6] and deletion in DJCHR [18, 19]. First we find all constraints that may have been affected by the retracted constraint from an internal storage of dependencies (like in [19]). Second we enlarge the affected variable domains to the maximum inferrable size (analogously to [6]). And finally we repost all relevant constraints such that a new reduction of variable domains is achieved (like in [19] and [6]). In contrast to the mentioned algorithms we do not make use of any global data structures in our algorithm. We use neither a central constraint store like [19] nor a centralized history trail like [6]. Omitting such central structures allows the concurrent execution of different solvers without further synchronisation. This is a prerequisite for distributed (non client-server) applications [15].

To specify the retraction algorithm in Fig. 4 we have to define, how the possibly affected constraints are deduced from the internal data structures.

Definition 8 (consequences, dependent and affected constraints). *Given an ACS constraint* $c = (\overline{v}, T, acs, s, E, \overline{h})$*, the set of consequences of* c *is given* *by* $E \subseteq \mathsf{cons}(c)$ *and* $c' \in \mathsf{cons}(c) \Rightarrow \mathsf{cons}(c') \subseteq \mathsf{cons}(c)$. *The set of dependent constraints is given by* $\forall v = (\overline{p}, (C_p)_{p \in P}) \in \varepsilon(\overline{v}) : \forall p \in P : C_p = \langle ..., c, c_1, ..., c_m \rangle \Rightarrow c_1, ..., c_m \in \mathsf{dep}(c)$ *and* $c' \in \mathsf{dep}(c) \Rightarrow \mathsf{dep}(c') \subseteq \mathsf{dep}(c)$. *The set of all affected constraints of* c *is given by* $\mathsf{aff}(c) := \mathsf{cons}(c) \cup \mathsf{dep}(c) \cup \bigcup_{c' \in \mathsf{dep}(c)} \mathsf{cons}(c')$.

foreach $c' \in \mathsf{builtin}(c)$ **do** // c' posted in c
  c'.`retract`() // retract c' also
end
foreach $c' = (\overline{v}', T', acs', s, E', \langle h_1, ..., h_n \rangle) \in \mathsf{aff}(c) \cup \{c\}$ **do**
  foreach $v_i = (\overline{p}, C) \in \varepsilon(\overline{v}')$ **do** // look at every variable
    $v_i := (\mathsf{dom}^{-1}(\mathsf{dom}(h_i) \cup \mathsf{dom}(\overline{p})), C)$ // enlarge variable domain to maximum
    $C := C \setminus \{c\}$ // remove c from dependent constraints
  end
  $E' := E' \setminus \{c\}$ // remove c from consequences
end // all affected variable domains enlarged to max history
foreach $c' = (\overline{v}', T', acs', active, E', \overline{h}') \in \mathsf{aff}(c)$ **do**
  c'.`post`(acs') //repost relevant, active constraints
end

Fig. 4. Execution of a constraint retraction c.`retract`() of the constraint c.

3 Semantics

In this section we define the denotational semantics for constraint programs (Def. 5) as the fixed point of the (infinite) Chaotic Iteration of domain reduction functions [1]. After that we show that the execution of constraint programs in ACS leads to this fixed point in finite time, i.e. the operational semantics is sound and complete and the evaluation terminates.

Every additive constraint task $t \in \mathsf{tasks}(c)$ defines a reduction of the domains of c's variables. This reduction depends on the domains of all $v \in \mathsf{vars}(c)$ and the implementation of the propagation task. Since the implementation is fixed during runtime the domain reduction can be considerd a function in the cartesian product of the initial domains $\mathsf{dom}^*(v)$ of all $v \in \mathsf{vars}(c)$. By canonically extending the domain and codomain of this function to the scheme of the program c was posted in we get a domain reduction function [1] as intended semantics of the constraint.

Proposition 1 (interpretation). *Given a constraint* $c = (\overline{v}, T, acs, s, E, \overline{h})$ *that is posted in a constraint program* P *with* $\mathsf{scheme}(P) = \langle v_1, ..., v_n \rangle$ *then the interpretation of* c *is defined by the mapping* $\mathsf{drf}_c(\langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n) \rangle) \mapsto \langle d'_1, ..., d'_n \rangle$

with $d'_i := \begin{cases} \mathsf{dom}(v_i) \cap \bigcap_{v_i.\mathtt{in}(\overline{p}) \in \mathsf{red}(c)} \mathsf{dom}(\overline{p}), & \text{if } v_i \in \varepsilon(\overline{v}) \\ \mathsf{dom}(v_i), & \text{else} \end{cases}$

The interpretation of a constraint is considered the maximum reduction of the variable domains that can be derived from the programmer's knowledge and the current variable domains.

Lemma 1. *Every additive ACS constraint in a constraint program P defines a set of monotonic domain reduction functions on* $\mathsf{scheme}(P)$.

Proof: According to Def. 2.2 and Def. 6.3 every ACS constraint c defines a set of constraint tasks $I = \mathsf{tasks}(c) \cup \bigcup_{c' \in \mathsf{builtin}(c)} \mathsf{tasks}(c')$. Since c is additive, the only influence that c may have on variable domains are, according to Def. 6, the domain reductions $\mathsf{red}(i)$ (Def. 6.1) defined for all $i \in I$. These reductions are monotonic wrt. set inclusion, because in Def. 6.1 we consider the set theoretical interpretation of the variable domains.

Due to Prop. 1 we stick to valid variable domains such that all $i \in I$ define domain reduction functions according to their definition in [1]. □

Given an additive constraint program P with $\mathsf{scheme}(P) = \langle v_1, ..., v_n \rangle$ and $\mathsf{constr}^+(P) := \mathsf{constr}(P) \cup \bigcup_{c \in \mathsf{constr}^+(P)} \mathsf{builtin}(c)$ we consider the set $\mathsf{drf}(P) := \bigcup_{c \in \mathsf{constr}^+(P)} \mathsf{drf}_c$ the domain reduction functions of P. The set $\mathsf{drf}(P)$ contains every domain reduction function that is the intrpretation of any constraint that was posted throughout the execution of P and is thus uniquely determined because of Lemma 1 and the algebraic properties of set union. Furthermore, $\mathsf{drf}(P)$ is finite iff the set $\mathsf{builtin}(c) \cup \bigcup_{c' \in \mathsf{builtin}(c)} \mathsf{builtin}(c')$ is finite for every $c \in \mathsf{constr}(P)$. This precondition has to be considered, because we allow, in contrast to [1], constraint postings as side-effects of the execution of constraint tasks (Def. 6.4).

Provided $\mathsf{drf}(P)$ is finite, we can derive from Apt's domain reduction theorem [1] that a fixed point of the Chaotic Iteration of $\mathsf{drf}(P)$ on $\mathsf{scheme}(P) = \langle \mathsf{dom}^*(v_1) \times ... \times \mathsf{dom}^*(v_n) \rangle$ exists and is uniquely determined. We call this fixed point the denotational semantics of P: $\mathsf{sem}(P)$.

3.1 Soundness and Completeness

Now we show that an ACS will evaluate variable domains that are equal to those in the denotational semantics. The state of an ACS program is denoted by the current variable domains and the solvers' buffers. If all buffers are empty, the state is consisdered a result of the program.

Definition 9 (state and result of a program). *The state of a constraint program P with* $\mathsf{solvers}(P) = \{(B_1, s_1, \rho_1), ..., (B_m, s_m, \rho_m)\}$ *and* $\mathsf{scheme}(P) = \langle v_1, ..., v_n \rangle$ *is given by* $(\langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n) \rangle, \{B_1, ..., B_m\})$. *The result* $\mathsf{res}(P)$ *of P is the state* $(\langle \mathsf{dom}(v_1), ..., \mathsf{dom}(v_n) \rangle, \{\langle\rangle\})$ *that is reached after the execution of P.*

Assuming that the considered constraint program is additive and has a result we can show that with propagation in ACS variable domains are restricted just as far as intended.

Lemma 2 (soundness and completeness of propagation). *For every additive constraint program P with result* $(\langle D_1, ..., D_n \rangle, \{\langle\rangle\})$ *the inclusion* $\mathsf{sem}(P) \subseteq D_1 \times ... \times D_n$ *(soundness) and the inclusion* $D_1 \times ... \times D_n \subseteq \mathsf{sem}(P)$ *(completeness) hold.*

Proof: The full proof can be obtained from the author. The idea of the proof of soundness is that we use the implemented domain reduction itself as intended semantics (Proposition 1). The idea for the proof of completeness is that every constraint task is executed at least once and that with every domain reduction we repost all depending constraints (Def. 6.1). $\square$

In order to make Lemma 2 applicable for non-additive constraint programs we show that our retraction algorithm leads to the same result as one would get, if the retracted constraint was never posted. We still assume that the result exists.

Lemma 3 (correctness of retraction). *Let P_1 be a constraint program with result $(\langle D_1, ..., D_n\rangle, \{\langle\rangle\})$ and $P_2 := P_1 \oplus \{c = \texttt{new Constraint}(\overline{v}, T), c.\texttt{post}(s), c.\texttt{retract}()\}$ another constraint program with the three mentioned additional statements. Then $\mathsf{res}(P_1) = \mathsf{res}(P_2)$ holds.*

Proof: The full proof can be obtained from the author. The idea is the following: The retraction can only be executed after the posting of c according to Def. 6, i.e. $\mathsf{res}(P_2)$ must exclude the semantics of c. For the evaluation of the result only the variables $V = \bigcup_{c' \in \mathsf{aff}(c)} \mathsf{vars}(c')$ must be considered, because all other variables keep their domains. For all $v \in V$ we restore the domains to the largest set that can be deduced from the history snapshots (Fig. 4, line 6). Due to the reposting of all $c' \in \mathsf{aff}(c)$ (Fig. 4, line 12) and Lemma 2 we get the desired state after the propagation is finished. $\square$

Now we state that ACS is an adequate means to solve CSPs that are implied by constraint programs on the basis of the implemented constraint tasks.

Theorem 1 (asynchronous constraint solving). *If a constraint program P has the result $\mathsf{res}(P) = (\langle D_1, ..., D_n\rangle, \{\langle\rangle\})$ the equation $\mathsf{sem}(P) = D_1 \times ... \times D_n$ holds.*

Proof: The existence of $\mathsf{sem}(P)$ is implied due to Theorem 2. The theorem follows from Lemma 2 and Lemma 3: We can generalize Lemma 3 such that multiple retractions are allowed in the program. Furthermore the order of the constraint postings and the order of variables in $\mathsf{scheme}(P)$ has no influence on the result of P. The full proof can be obtained from the author. $\square$

3.2 Termination

First we show that a constraint program will terminate if the programmer provided well-defined constraints.

Lemma 4 (termination). *Every constraint program P has a result, if for all $c \in \mathsf{constr}(P)$ the following conditions are satisfied:*

(1) all $t \in \mathsf{tasks}(c) \cup \bigcup_{c' \in \mathsf{builtin}(c)} \mathsf{tasks}(c')$ terminate

(2) $\mathsf{builtin}(c) \cup \bigcup_{c' \in \mathsf{builtin}(c)} \mathsf{builtin}(c')$ *is a finite set.*

Proof: The full proof can be obtained from the author. Considering condition 2 we know that only a finite set of tasks can be inserted in the buffers for their first execution. We can show that the retraction algorithm (Fig. 4) terminates by showing that the set $\mathsf{aff}(c)$ is finite for every c. Together with condition 1 this means that every task that is executed in a solver terminates. Furthermore we can show that every task cannot be inserted in a buffer infinitely often. This is deduced from the fact that the domain reductions defined in the tasks are idempotent.
□

In theorem 1 we suppose that the denotational semantics of a program is well defined. This can be derived from the fact that the semantics exist if a result exists.

Theorem 2. *If a constraint program P has a result then* $\mathsf{sem}(P)$ *is well defined.*

Proof: Follows from condition 2 in Lemma 4, Lemma 1 and the domain reduction theorem [1].
□

4 Implementation and Performance Results

A prototypical solver called J.CP that uses the ACS paradigm has been implemented in Java. Beside the generic solver-package itself we provide packages for finite integer domain contraint solving and for finite sets. J.CP provides a generic implementation of variables that can be parameterized by any type of domain. Every variable can be used either as local or as remote variable, in the latter case it can be accessed via Internet from any other J.CP solver or constraint.

Constraint tasks are executed in a local solver that runs in a separate thread. As we already mentioned, J.CP allows the evaluation of hyper-arc-consistency (Def. 6.1), however, the achievement of certain reliability-values in the solver can be interpreted as node-, arc- , 3- or more-than-3-consistency. The scheduling takes care, that tasks of n-ary constraints are scheduled as $(n-1)$-ary constraints as soon as one of its variables is instantiated.

New constraints can easily be integrated in the system by implementing our constraint interface. With new constraints and new implementations of variable domains we can thus integrate new constraint systems into J.CP without making any conceptional changes. Furthermore even heterogeneous CSPs can be solved if the repective constraints over variables with different domain-classes are implemented (e.g. `Leq(FDVar x, RealVar y)`). We do need any extra communication interface like in [2], but can implement the intended interaction of variables of different systems based on domain-knowledge of the constraint-systems using basic techniques such as type casting. For example, the mathematical relation $\leq$ can be interpreted by the mentioned `Leq`-Constraint over an integer-domain-variable `x` and real-domain-variable `y` by interpreting the integer values of `x` as real values. We think, that for every heterogeneous constraint, that has a declarative semantics there is such an interpretation of one variable in the domain of the other.

If a constraint task or a variable detects an inconsistency, it causes an inconsistency event that is broadcast in the system. Inconsistency events are handled by Java event listeners, that can be considered implementations of ACS repair-operations (Def. 3). The handlers are supposed to analyze the failure and to retract and repost constraints, making a solution possible. Such event listeners are used to implement non-deterministic search algorithms (labeling-constraints), where several retractions and postings of variable-instantiation constraints are performed.

4.1 The Influence of Constraint Scheduling

As already mentioned in Remark 1, the buffers of asynchronous constraint solvers allow the scheduling of constraint tasks. Due to the asynchronous execution we can not only schedule the tasks within the propagation of one constraint like in ECL^{i}PS^e [17] but are able to collect and schedule the tasks of multiple posted constraints and their propagation.

With expert knowledge on constraint propagation such scheduling can significantly reduce the computation time of the solver as shown in Table 1.

Table 1. Comparison of J.CP in synchronous and asynchronous mode with different task buffers and SICStus [3] v3.8.7 on a dual Pentium III-500 Linux PC. All numbers are average values (in milliseconds) of 10 successful tests. In 90 runs of SICStus we had 7 memory allocation errors, such that extra runs were performed. All values are given in milliseconds.

	J.CP	J.CP async.				SICStus
	sync.	queue	stack	set	special	(jasper)
`order 200`	2867	1868	1785	1952	449	1069
`order 500`	15292	9217	7752	8541	526	1969
`order 1000`	76711	34818	23872	24876	1024	6933
`reorder 1000 1`	110342	35208	24001	25112	981	10573
`reorder 1000 500`	103807	17900	11644	15840	920	7389
`reorder 1000 999`	103283	34259	27461	28732	1003	6944

The `order n`-benchmark can be described as follows $\forall x_i \in \{x_1, x_2, ..., x_{\mathtt{n}}\}$: $x_i \in \{1, 2, ..., \mathtt{n}\} \wedge x_{i-1} < x_i$. The only solution is thus $x_i = i$ for all $i \in \{1, 2, ..., \mathtt{n}\}$, which can be found without any search using the implementation of $<$ shown in Figure 2. We chose this benchmark, that knowingly does not require any sophisticated constraint implementation to test the execution model unaffected from propagation or search algorithms. In `reorder n m`, we retract the constraint $x_m < x_{m+1}$ after the execution of `order n`. In the first column we use immediate execution of every task like it is done in most CLP languages. In the asynchronous execution (columns 2-5) we distinguish different buffer types used in the solver. In the column marked with 'special' we used a stack-buffer and a special implementation of $<$ that propagates all lower bounds of the variable domains before the upper bounds are propagated. In the last column we show the execution time of SICStus solver with its Java interface jasper, where retraction is performed by backtracking to the named constraint and reposting all constraints that were subsequently called.

In the `reorder` benchmarks and in the column marked with 'J.CP special' we use different priorities of constraint tasks to improve the propagation speed. As expected we could achieve a massive speedup by scheduling the retraction task before all propagation is finished. But also in the `order` benchmark we could reduce the time complexity from $O(2n^2)$ to $O(n)$ with the usage of expert knowledge on constraint propagation (column marked with "special").

4.2 Encapsulated Search for ACS

As mentioned in the preliminaries, we consider finding a solution to a CSP C the same as finding instantiation constraints for all variables in C such that no inconsistencies occur within their propagation. An instantiation constraint over a variable binds the variable to a value and thus causes propagation. It fails, if the value is not in the domain of the variable. In J.CP we implement encapsulated search algorithms in constraints that post and retract instantiation constraints until no more inconsistencies are detected [14]. Since constraint implementations are imperative programs we can use any algorithm here. With J.CP we provide two generic search algorithms (backtracking and simulated annealing), that can be used in heterogeneous or distributed CSPs. We can use either of these constraints to find instantiations of variables with different kinds of domains, because in the search algorithms we refer to abstract interfaces of variables and instantiations. The implemented algorithms can (as any J.CP constraint) also be used if the variables are distributed among the Internet. To evaluate our first implementations of search algorithms we ran the famous n-queens benchmark (implemented with $\neq$-constraints) as shown in Table 2.

Table 2. Comparison of J.CP using backtracking (with fixed variable order and random value selection) and simulated annealing to other Java constraint solvers. JCL 2.1 [8] is used with its 'forward checking arc consistency' solver, JaCK [10] uses standard backtracking and SICStus/jasper [3] v3.8.7 was tested with and without the "first fail" labeling strategy. All numbers are average values (in milliseconds) of 3-10 tests, ran on a dual Pentium III-500 Linux PC.

	J.CP BT	J.CP SA	JCL	JaCK	SICStus	SICStus ff
`5 queens`	308	307	49	1061	739	736
`8 queens`	323	312	23	7814	768	760
`12 queens`	913	556	29	30274	835	828
`20 queens`	3617	2015	18575	$>$ 5 min	12228	1061
`50 queens`	9326	4925	$>$ 5 min	$>$ 5 min	$>$ 5 min	4968
`100 queens`	20934	12342	$>$ 5 min	$>$ 5 min	$>$ 5 min	22123

The runtime of J.CP is competitive with the professional solver SICStus and could still be improved by introducing the first fail strategy into backtracking for example. The tests have shown that local search is faster in most cases, which could be expected in this problem. However, the variance was much higher than with backtracking. Furthermore simulated annealing was slower in other benchmarks that had a smaller quotient of solutions per possible assignments.

5 Conclusion and Future Work

We have defined an execution model for constraint propagation in concurrent object-oriented programs. Integrating the possibilities of Constraint Satisfaction, our paradigm is well suited for interactive constraint-based software. Constraints can be posted or retracted in concurrently running, cooperating solvers at any time, while the (intermediate) result of constraint propagation or search algorithms can be read from constraint variables. This allows distributed applications, where the users can interactively and cooperatively define a CSP based on the consequences of their earlier decisions which is represented in terms of the possible values for uninstantiated variables. The solvers, running in parallel to the application, can use the spare time that emerges during user interaction for expensive, non-standard constraint propagation. With the scheduling of unprocessed tasks we could achieve a speedup compared to synchronous execution that often evens up the overhead of the task buffering and distributed history management.

In future work, we will concentrate on the integration and improvement of generic search algorithms (as labeling-constraints) in the system. With the possibility of constraint retraction, there is a great deal of potential with the combination of incremental constraint solving and Local Search.

References

[1] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1998.

[2] Franz Baader and Klaus U. Schulz. Combination of constraint solvers for free and quasi-free structures. *Theoretical Computer Science*, 192:107–161, 1998.

[3] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Proc. PLILP97*, volume 1292 of *LNCS*, 1997.

[4] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In *Proc. CP01*, LNCS 2239, pages 240–255. Springer, 2001.

[5] The Mozart Consortium. `http://www.mozart-oz.org`.

[6] Yan Georget, Philippe Codognet, and Francesca Rossi. Constraint retraction in CLP(FD). *Constraints*, 4(1):5–42, February 1999. ISSN 1383-7133.

[7] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) system and language. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[8] EPFL Lausanne. JCL. `http://liawww.epfl.ch/~akira/JCL/`.

[9] Irvin Lustig. Optimization and Java. *Java Developer's Journal*, 6(6), 2002. `http://www.sys-con.com/java/`.

[10] LMU Munich. JaCK, `http://www.fast.de/~mandel/jack/jack.html`.

[11] Nat. Univ. of Singapore. Figaro library, `http://figaro.comp.nus.edu.sg`.

[12] OCRE Project. Choco web pages, 2002. `http://www.choco-constraints.net`.

[13] ILOG S.A. `http://www.ilog.fr`.

[14] Georg Ringwelski. Search in Asynchronous Constraint Solving CP'02 Doctoral Programme `http://www.math.unipd.it/~frossi/ringwelski.ps`.

[15] Georg Ringwelski and Hans Schlenker. Distributed Constraint Satisfaction with Asynchronous Solvers. *ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming*, Cork, 2002.

[16] Christian Schulte. Programming constraint inference engines. In *Proc. CP97*, LNCS 1330, pages 519–533. Springer, 1997.

[17] M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997.

[18] Armin Wolf. Adaptive Constraint Handling with CHR in Java. In Toby Walsh, editor, *Proc. CP2001*, LNCS 2239, pages 256–270. Springer, 2001.

[19] Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On incremental adaption of constraint handling rule derivation after constraint deletions. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, 2000.