An Asynchronous Execution Model for Constraint Processing in Object-Oriented Software

Georg Ringwelski

GMD FIRST, German National Research Center for Information Technology
Kekuléstraße 7, 12489 Berlin, Germany
Email: georg.ringwelski@gmd.de

Abstract. In this paper we define an object-oriented system that can be used to solve constraint satisfaction problems by incremental propagation and search. Constraints can be posted or retracted asynchronously in one or more solvers running concurrently to the application. The semantics of a constraint program is given by a fixed point of the domain reductions that are caused by the implementations of all posted constraints. This system does not make use of any global information and is thus well suited for distributed constraint processing. We present a prototypical implementation of such an asynchronous constraint solver together with some runtime tests.

1 Introduction

Constraint programming (CP for short) has reached many application areas of Artificial Intelligence since it was introduced in the early 1980's. Based on the sound theoretical basis of constraint logic programming (in short CLP)[9, 8] or concurrent constraint programming [12], it was integrated in many programming languages, most of them being Prolog implementations such as clp(R) [9] or SICStus Prolog [4]. The interest in the integration of constraint processing into non-logic programming languages has grown together with the acceptance of CP in the software community and industry. Thus some systems have been implemented to integrate CP into object-oriented programming languages such as C++ [11] or Java [5, 3]. The most popular development of this approach is ILOG Solver (`www.ilog.fr`) which is cannot be considered a CLP system anymore.

1.1 Motivation

The success of constraint programming results to a large extend from the efficiency of the used algorithms. Restricting variable domains with cooperating propagation and search seems to be a very powerful concept to find solutions for combinatorial problems. However, the use of CLP did only sparsely cross the border of academic applications. We think, this is (among others) caused by the method of software development that is used in most cases with CP. Because of the uncertainty of the applicability of constraint-based methods Rapid Prototyping often is used. This, however, hardly allows the implementation of "professional size" applications. Additionally the integration of CLP into modern systems, that make use of much communication beetween different program components often brings up non-trivial problems. Much research on the use of CP in distributed systems has been done in the recent years [13]. The ability to react on user-interaction also requires extensions to the CLP scheme. Not only normal input and output, but also constraint retraction [7] have to be "squeezed" into the logic programming paradigm.

Our idea is to use the efficient problem solving of CLP, i.e. algorithms for propagation and search, in a programming environment that is better suited for the use in laarge, heterogeneous and interactive software systems. We decided to define an execution model for constraint processing in imperative programs that allows distribution and interaction. The (possibly parallel running) constraint solvers are used as information servers, while in client applications constraints can be posted and retracted asynchronously. The result of the solvers' concurrently executed evaluation can be read in the constraint-variable objects. The design of the system is purely object-oriented, such that modern software development tools and methods can be used for the professional development of software systems that use constraint-based methods. We hope, an implementation of our concept will allow the integration of CP into large software systems thus yielding a wider spread of constraint technology.

1.2 Preliminaries

A Constraint Satisfaction Problem (in short CSP) $P = (X, C)$ is given by a set $X = \{(x_1, d_1), ..., (x_n, d_n)\}$ of variables x_i with associated value sets d_i and a set C of constraints over X. The elements of C restrict the sets d_i that are called the domain of the variables x_i. The objective of constraint propagation is to transfer a CSP to another CSP, that is "easier". In our context this means, that the domains of variables are restricted (refinement search). A CSP can be defined by a finite stream of constraint postings and retractions, that iteratively build a constraint-network [10].

In sections 4 and 5 we make use of some general notions on the iteration process of constraint propagation. To keep the paper self-contained we recapitulate some definitions and a theorem from [1]: In a partial order $(D, \sqsubseteq)$ a function $f : D \to D$ is called inflationary, iff $\forall x \in D.x \sqsubseteq f(x)$ holds. It is called monotonic, iff $\forall x, y \in D.x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ holds. Given a set of functions $F = \{f_1, ..., f_n\}$ on D, and a value $d \in D$, then the chaotic iteration of F is a sequence $\langle d_0 := d, d_1, d_2 ... \rangle$, with $d_j := f_{i_j}(d_{j-1})$, where every f_i is called infinitely often. A structure $(D, \sqsubseteq)$ is called a $\sqcup$-po, iff D contains a least element $\perp$ and for every sequence $d_1 \sqsubseteq d_2 \sqsubseteq ...$ the least upper bound $\bigsqcup_{n=0}^{\infty} d_n$ of the set $\{d_1, d_2, ...\} \subseteq D$ and finally for all $a, b \in D$ the least upper bound $a \sqcup b$. Since we use sets D as variable domains we can use the structure $(\mathcal{P}ow(D), \supseteq)$ as $\sqcup$-po for chaotic iteration.

Theorem 1 (Domain Reduction). *Given a CSP* $P = (\{(x_1, D_1)..., (x_n, D_n)\}, C)$ *such that every* $c_i \in C$ *defines an inflationary and monotonic function* f_i *in* $\mathcal{P}ow(D_1) \times ..., \times \mathcal{P}ow(D_n)$. *Then the limit of every chaotic iteration of* $F := \{f_1, ..., f_m\}$ *exists and this limit coincides with* $\bigcap_{j=0}^{\infty} f^j(D_1, ...D_n)$ *for every* $f \in F$.

2 Architecture and Application

An Asynchronous Constraint Solver (in short ACS) system, presented in Figure 1, consists of five main classes. A **constraint program** (Def. 4) that defines a CSP through the definition of variables and constraints, that are posted or retracted in defined solvers. Such programs may be distributed or heterogeneous applications that may react on user interaction by constraint postings or retractions thus changing the CSP being processed in an incremental manner. **Variables** are associated to a domain, which represents the values, the variable may have. We do not need to determine a special kind of domain, such as FD for example, here. The strict typing of imperative languages allows any object to be handled by the associated constraints if this is intended. If a variable or its domain cannot be handled by a special constraint the compiler will notify the programmer. **Constraints** define algorithms that may restrict the domains of their variables. These algorithms are executed as solver-tasks if the constraint is explicitely posted in a solver or during a propagation process. If this algorithm restricts the domain of any variable to the empty set, i.e. there are no values left that the variable could have, an inconsistency occurred. This causes an inconsistency-event that is broadcasted in the system. Inconsistency-events are handled by **event listeners**, that are defined in the system. They are to analyze the failure and to retract and repost constraints, such that a solution will exist. Such event listeners are used to implement non-deterministic search algorithms, where several retractions and postings of variable-instantiation-constraints are performed. **Solvers** run in concurrent threads, scheduling and calling the tasks defined in the constraints that were posted in them. A constraint cannot be posted in more than one solver.

The architecture is designed for the integration of constraint processing into complex software systems supporting modern features as described in the following. ACS can easily be used in **distributed systems**, because in difference to CLP it does not make use of any global information. Solvers running on different machines, can process local information, while only the necessary information is communicated via common variables. Constraint-postings from different agents can be processed by the solvers, because this is performed asynchronously anyway. ACS can easily be used in highly **interactive systems**, because in most imperative programming languages manifold features for input and output are supported. The thus received information can be processed directly by the solver, because incremental constraint posting and retraction are supported. These features allow the (imperative-language-)programmer to develop constraint-based programs without understanding the execution model of CLP. Constraint-based

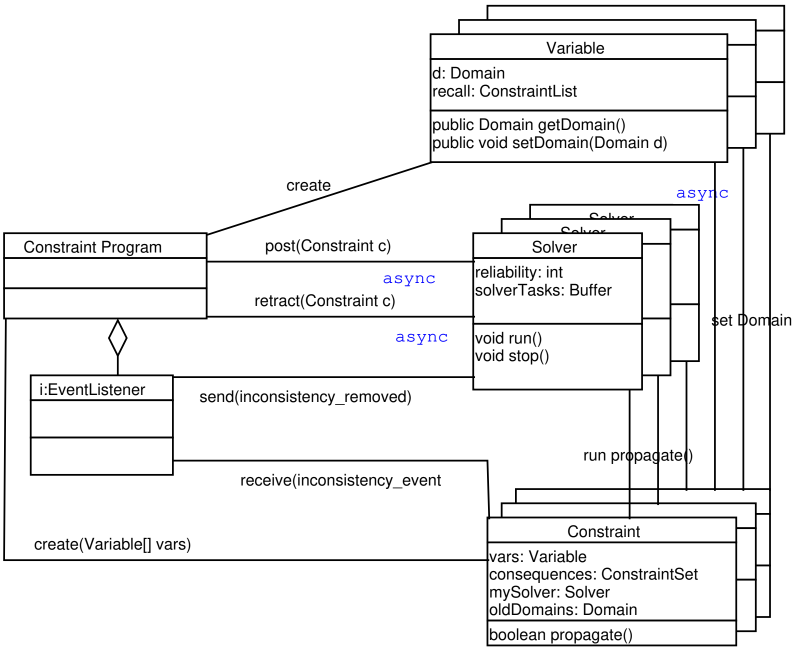

Fig. 1. Architecture of an Asynchronous Constraint Solver.

methods can be integrated in professional **software development processes** because everything is implemented in regular objects. ACS can be extended with domain specific constraints and variable domains through the implementation of the desired classes. A composition of constraint systems [2] is performed in a generic way, using the type-system of the programming languages. The analysis and design of constraint-based programs can be made in software-engineering tools making use of established methods for software development such as design-patterns or UML.

3 Static Components of ACS

Constraint variables are associated to a representation of their domain. This representation is given in a vector of features (properties) non-ambiguously defining a set. For example the set of integer values contained in a finite interval can be defined by the maximum and the minimum value of the interval, such that this set could be defined by the properties `minValue:int` and `maxValue:int` (see also [6]). If such a feature is changed, constraints depending on it must be reposted (propagation) in order to reach arc-consistency in the constraint network.

Definition 1 (Constraint Variable). *A (constraint-) variable* $v = (\overline{p}, C)$ *is given by a vector* $\overline{p} \in P = P_1 \times P_2 \times ... \times P_n$ *of properties of the domain* D

of v*, such that a bijective mapping* $dom : P \to \mathcal{P}ow(D)$ *exists, and a family* $C = (C_p)_{p \in \{P_1,...,P_n\}}$ *of sequences of constraints* C_p*.*

Every constraint holds information on its variables and the solver it was posted in. Furthermore the consequences and the history of the constraint are stored, which is necessary information for constraint retraction.

Definition 2 (Constraint). *A constraint* $c = (\overline{v}, i, s, C, H)$ *is given by a vector* $\overline{v}$ *of its variables, an implementation* i *(which is defined in Def. 6), its solver* s*, a set* C *of constraints called* c*'s consequences, and a* $\overline{v}$*-indexed family of sets (variable domains)* $H = (h_{v_i})_{v_i \in \overline{v}}$ *called* c*'s history.*

The execution of the implementation given with every constraint is organized by a constraint solver which is running asynchronously in our approach. The solver always provides information about 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.

Definition 3 (Asynchronous Constraint Solver). *An asynchronous constraint solver* $s = (B, r)$ *is given by a task-buffer* $B = (T, next)$ *consisting of a set* $T = \{post(c), retract(c) \mid c = (\overline{v}, i, s, C, H)\}$ *of solver-tasks and a scheduling* $next :\to T$*. Furthermore a reliability value* $r \in \{0, 1, ..., \top\}$ *is given with every solver.*

4 Execution of Constraint Programs

Provided an ACS and implementations of the required constraints, any CSP can be formulated in an imperative program by defining variables and posting constraints over them in declared solvers. An example Java implementation of a constraint program is presented in Figure 5

Definition 4 (Constraint Program). *A constraint program is an imperative program using the following primitives*

- *solver declarations:* $s = newSolver()$*,*
- *variable definitions:* $v.set\_domain(\overline{p})$ *for previously undefined variables* v*,*
- *domain queries:* $d = v.get\_domain()$ *for defined variables* $v = (\overline{p}, C)$*,*
- *constraint postings:* $s.post(c)$ *of a constraint* $c = (\overline{v}, i, s, \emptyset, \emptyset)$ *in a declared solver* s*, with defined variables* $\overline{v}$
- *constraint retractions:* $s.retract(c)$ *in a declared solver* s*.*

Where v *is a variable,* $\overline{p}$ *describes its domain* d *(*$dom(\overline{p}) = d$*) and* c *is a constraint.*

Any desired constraint can be integrated into a given system, by implementing some domain specific knowledge into a propagation algorithm. This algorithm will be executed by the solver, such that the farest possible domain reduction is achieved. The aim is, that this reduction will lead to a solution of the CSP that is implemented in the application program. Normally, however, nondeterministic search, i.e. an algorithm, that coordinates postings and retractions of instantiation-constraints, is necessary for this.

Definition 5 (Result of a Constraint Program). *Consider a constraint program P defining the solvers S and the variables V. If the reliability of all solvers in S has reached $\top$ after the execution of P, then the result of P is defined as the conjunction $\bigwedge_{v_i \in V} v_i \in v_i.get\_domain()$.*

Please notice, that not every constraint program has a result. It is possible, that due to repeated constraint postings top-reliability is never reached. This may be the case, if in any endless program loop constraints are posted, or if in any constraints non-inflationary recursive calls of constraint postings are implemented.

4.1 Declaring Solvers

A solver declaration $(B, r) = newSolver()$ starts the concurrently running solver-thread which executes all posted propagation or retraction tasks. If a solver-task fails, it broadcasts an $inconsistency\_event$, that should be processed by the application by retracting and reposting constraints. If this is finished, the application sends a signal to the solving process to continue. If the task-buffer is empty, the solver waits until a new task is to be processed. As long, as constraints were posted or retracted in the solver they will be scheduled by the function $next()$. With this scheduling we make use of different priorities for different solver-tasks. Constraint retractions have the highest priority, because the earlier they are performed, the less they will interfere with propagation and thus the less changes have to be reversed when the retraction is executed. Another possibility arising here is to implement several tasks with different priorities in one constraint. Thus the execution of expensive propagation algorithms may be delayed until the solver has some spare-time while important propagation algorithms are executed early. All tasks are processed by the function

$$execute(task) = \begin{cases} run(impl), & \text{if } task = post((\overline{v}, impl, s, C, H)), \\ remove(c), & \text{if } task = retract(c), \end{cases} \tag{1}$$

which will be described in Sections 4.3 and 4.4.

4.2 Defining Variables

A variable definition $v.set\_domain(\overline{p}')$ is processed in a variable v by the execution of the program shown in Figure 2. If a variable definition restricts the set of possible values for a variable, its domain is restricted to the intersection of the former and the newly defined set. Furthermore, propagation is started, to provide the new information for all constraints, that might find further restrictions.

4.3 Constraint Posting

The posting of a constraint $solver.post(c)$ is processed by storing the current variable domains in the history of the posted constraint, if this is not already

```
1 Cons := new ConstraintSet();                    //create empty set of consequences
2 foreach p'_i ∈ p̄' do                           //set property, so that domain is reduced
3   if dom(p_1, ..., p'_i, ..., p_n) ⊂ dom(p_1, ..., p_i, ..., p_n)
4     then p_i := p'_i;                                           //reduction of the domain
5          foreach c = (v̄, i, s, C, H) ∈ C_{P_i} with c.isValid() do
6            s.post(c);                                                   //propagation
7            Cons := Cons ∪ {c}                                    //store consequences
8          end fi end
9 return Cons;
```

Fig. 2. Execution of a variable-definition $v.set\_domain(\overline{p}')$ in a variable $v = ((p_1, ..., p_n), (C_{P_1}, ..., C_{P_n}))$. The call of $c.isValid$ returns true, iff c has not been retracted.

defined. After that the implementation is added to the task buffer of the corresponding solver.

The execution of a propagation-task $run(impl)$ (see (1)) is performed by the execution of the constraint's implementation $impl$. An example Java implementation of a constraint is presented in Figure 4

Definition 6 (Constraint Implementation). *The implementation i of a constraint $c = (\overline{v}, i, s, C, H)$ is an imperative program using the following primitives*

1. *variable definitions: $cons := v.set\_domain((p'_1, ..., p'_n))$ for variables $v = ((p_1, ..., p_n), C_P) \in \overline{v}$ such that*
   (a) *for all properties P_i with $p'_i \neq p_i$ the last element of the sequence $C_{P_i} \in C_P$ is c.*
   (b) *the constraints, that were reposted by v are added to the known consequences: $cons \subseteq C$*
2. *domain queries: $v.get\_domain()$ for $v \in \overline{v}$*
3. *constraint postings: $s.post(c')$, with $c' \in C$,*
4. *event raisings: $broadcast(inconsistency\_event)$*

Such implementations define functions on the cartesian product of all domains in a program that can be used for chaotic iteration. They are defined by the intersection of the former domains and the domains described in variable-definitions.

Definition 7 (Domain Reduction Function). *Consider a constraint program that defines the variables $V = \{v_1 = (\overline{p}_1, C_1), ..., v_n = (\overline{p}_n, C_n)\}$, and posts a constraint $c = (\overline{v}, impl, s, C, H)$ where impl contains variable- definitions $\{w_i.set\_domain(\overline{q}_i) \mid w_i \in \overline{w} \subseteq \overline{v} \subseteq V\}$. Then c defines a domain reduction function f in $\mathcal{P}ow(dom(\overline{p}_1)) \times ... \times \mathcal{P}ow(dom(\overline{p}_n))$:*

$$f(dom(\overline{p}_1), ..., dom(\overline{p}_n)) := (d'_1, ..., d'_n) \text{ where } d'_i = \begin{cases} dom(\overline{p}_i), & \text{if } v_i \notin \overline{w}, \\ dom(\overline{p}_i) \cap dom(\overline{q}_i), & \text{else.} \end{cases}$$

Due to Definitions 7 and 6 and to the lines 3 and 4 in Figure 2, it is clear, that every constraint implementation defines a domain reduction function.

Lemma 1. *Every domain reduction function is inflationary and monotonic wrt. to the partial order* $(s_1, ..., s_n) \sqsubseteq (s'_1, ..., s'_n) \Leftrightarrow s_1 \supseteq s'_1 \wedge ... \wedge s_n \supseteq s'_n$.

Proof: Given a domain reduction function f, with $f((d_1, ..., d_n)) \mapsto (d'_1, ..., d'_n)$ and $f((e_1, ..., e_n)) \mapsto (e'_1, ..., e'_n)$

1. For every $i \in \{1, ..., n\} d_i \supseteq d'_i$ holds because of Definition 7, thus f is inflationary wrt. $\sqsubseteq$, i.e. $(d_1, ..., d_n) \sqsubseteq f((d_1, ..., d_n))$
2. Let $(d_1, ..., d_n) \sqsubseteq (e_1, ..., e_n)$ be two vectors of domains of the variables $V = \{v_1, ..., v_n\}$ defined in a constraint program. Thus $e_1 \subseteq d_1 \wedge ... \wedge e_n \subseteq d_n$ holds. Let f be the domain reduction function defined by a constraint $c = (\overline{v}, impl, s, C, H)$; then the following cases are possible:
   Case $v_i \in V \wedge v_i \notin \overline{v}$: Because of Def. 7 $d'_i = d_i \wedge e'_i = e_i$ hold and thus $e'_i = e_i \subseteq d_i = d'_i$.
   Case $v_i \in V \wedge v_i \in \overline{v}$: then a variable definition $v_i.set\_domain(\overline{q})$ exists in $impl$. Thus, according to Def. 7 $d'_i = d_i \cap \mathsf{dom}(\overline{q})$ and $e'_i = e_i \cap \mathsf{dom}(\overline{q})$ hold. Because of $e_i \subseteq d_i$ and the latter equation we know that $e'_i \subseteq d_i \cap \mathsf{dom}(\overline{q})$, so that from the first equation it follows that $e'_i \subseteq d'_i$.
   Thus, in any case, $e'_i \subseteq d'_i$ holds for $i \in \{1, ..., n\}$ so that $(d'_1, ..., d'_n) \sqsubseteq (e'_1, ..., e'_n)$; but this means that f is monotonic.

■

4.4 Constraint Retraction

Recapitulating the previous section, the call of $s.post(c)$ of a constraint $c = (\overline{v}, i, s, C, H)$ may cause three changes in the system:

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

If c is retracted, these changes must be reversed in the system.

If a retraction-task $s.retract(c)$ is posted in $s = (B, r)$, it is added to the task-buffer of $s := (B \cup \{retract(c)\}, decrease(r))$. Once it is scheduled, its corresponding task $remove(c)$ (see (1)) is executed. The constraint c is removed from the constraint-network by a flag, signalizing, that c is not valid. If this flag is set, the constraint will not be reposted (Figure 2, line 5). In order to restore the three changes listed above, the implementation of $remove(c)$, shown in Figure 3, is executed with highest priority in s. The method *getAllConsequences* is provided by every constraint and returns a set of all consequences that the constraint had wrt. propagation. This set can be constructed as a union of direct consequences (Def. 6 (1b)) and the constraints that were reposted due to domain restrictions (Figure 2, line 7). The latter are collected in all affected constraints with calls of *getAllConsequences* on reached constraints.

The method *getAllLaterPostings* returns all constraints, that were first posted after the constraint that is to be deleted. These constraints can be reached

1 $c.setValid(false);$ //c is removed from the constraint network
2 $R := new\ ConstraintSet();$ // create empty set
3 $R := c.getAllConsequences(\emptyset);$ // get all consequences (2)
4 $R := R \cup c.getAllLaterPostings(\emptyset);$ // get later calls (3)
5 **foreach** $v = (\overline{p}, C) \in \overline{v}$ with $(\overline{v}, i, s, C, H) \in R$ **do**
6 $\quad v := (dom^{-1}(h_v), C \setminus \{c\});$ // restore domains (1)
7 **end**
8 $R := R \setminus \{c\};$
9 **foreach** $c = (\overline{v}, i, s, Con, H) \in R$ **do** //repost constraints in their solver
10 $\quad s.post(c);$
11 **end**

Fig. 3. Execution of a constraint-retraction $s.retract(c)$ with the method $remove(c)$ of the a constraint $c = (\overline{v}, i, s, Con, (h_{v_i})_{v_i \in \overline{v}})$ in a solver $s = (B, r)$. In line 6 the former domain can be evalueted, because dom is defined to be bijective.

in the constraint network via common variables, that all store their constraints in a list ordered by the first invocation. We do not need to store this information globally, because every constraint knows its variables and every variable knows its constraints and their order.

5 Semantics of ACS

If a CSP is implemented in a constraint program, the most extensive restriction to the variables' domains is considered the semantics of the program. In the best case this is a solution of the CSP. The desired final computation state of domain reduction is reached, once no further domain reductions can be derived from the constraints' implementations. The implementations can thus be considered transformations of the system yielding a supremum representing the final state. These transformations have already been formalized in definition 7 with the introduction of the domain reduction functions, that yield a supremum in a partial order defined by the variables' domains. Thus, we define denotational semantics of a constraint program as the fixpoint of the chaotic iteration of domain reduction functions of the posted constraints, i.e. the program state after completed propagation.

Definition 8 (State of Constraint Program). *The state $\{IS \wedge \bigwedge x_i \in D_i\}$ of a constraint program P is a conjunction of the state of the imperative program IS and a domain-information $x_i \in D_i$ for every variable x_i that is defined in P.*

Definition 9 (Denotational Semantics of Constraint Program). *Let a constraint program $Prog = \{S, D, P, R\}^+$ be a sequence of solver-declarations S, variable-definitions $D = \{v_i.set\_domain(\overline{p}_i) \mid i \in \{1, ..., n\}\}$, constraint-postings P and constraint-retractions R. Let further be C be the set of constraints that were posted in P and not retracted later. If after program execution* *for all* $(T, r) \in S$ $r = \top$ *holds, then the semantics of Prog is the transformation* $\{IS\} \mapsto \{IS^+ \wedge \bigwedge v_i \in dom(\overline{p}_i) \cap d_i\}$. *Where* d_i *is projected from the limit* $(d_1, ..., d_n)$ *of the chaotic iteration of the domain reduction functions defined by the implementations of every* $c \in C$.

Since according to Lemma 1 the implementation of a constraint (Def. 6) defines a (inflationary and monotonic) domain reduction function, Apt's domain reduction theorem [1] is applicable for the unretracted constraints C. All preconditions are satisfied, so that a chaotic iteration is well-defined and the existence and singularity of such a limit $(d_1, ..., d_n)$ is proved by [1].

Theorem 2 (Soundness and Completeness of ACS). *Given a constraint program* P *defining the variables* V *that has the semantics* $\{IS\} \mapsto \{IS^+ \wedge \bigwedge_{v_i \in V} v_i \in d_i\}$. *Then the result of* P *is* $\bigwedge_{v_i \in V} v_i \in d_i$.

A formal proof of the theorem can be obtained from the author. With analysis of the used algorithms of the execution model we could proof soundness and completeness of ACS wrt. the denotational semantics of a constraint program. Furthermore we proved, that constraint retraction does not interfere with these properties.

Provided, a program has a result, i.e. all tasks have finished their execution, every constraint program will lead to the same limit as the infinite chaotic iteration process.

6 Implementation and Performance Results

A prototypical solver for finite domain constraints that uses the asynchronous constraint solving paradigm has been implemented (according to Figure 1). It is a regular Java package called **J.CP** that can be used by any Java-application and runs on almost any hardware. Constraints are implemented in classes, so that new constraints can easily be integrated into the system. Every new constraint is a specialization of a given class, that provides all necessary methods for the execution in the ACS scheme. Thus only propagation algorithms and their priority have to be defined with every new constraint. The algorithm must include notifications of all variables, on which changes of their domain the constraint is to be reposted (Def. 6.1a). In Figure 4 we present as an example the implemetation of the "less-than" constraint in the finite domain constraint system The system is also extendable with other constraint systems, which is easy to handle because of the strict typing of Java.

In Figure 6 we show some results of runtime experiments of **J.CP** with synchronous and asynchronous constraint execution. In the first, the propagation algorithm of every posted or reposted constraint is executed immediately, whereas in the second case constraints are executed asynchronously like it was described in the previous sections. In the third column we present the results of SICStus Prolog [4] with its Java-interface jasper. However, the results cannot really be

```
public class Lt extends Constraint {
    public Lt(FDVar a, FDVar b){
        tasks[0] = new Task(this,new Object[]{a,b},6); // priority 6
    }
    Boolean propagate(FDVar a, FDVar b){
            // evaluate new domains
        int newATop = b.getMax() - 1 ; int newBBot = a.getMin() + 1;
        if(newATop < a.getMin() || newBBot > b.getMax())
            return new Boolean(false);  // failure detected
        else{
            consequences.addAll(a.setMax(newATop)); //change property...
            consequences.addAll(b.setMin(newBBot)); // ... and propagate
            a.recallOnMin(this); b.recallOnMax(this); // notify variable
            return new Boolean(true);  // successful execution
}   }   }
```

Fig. 4. Implementation of the "less-than" over finite domain constraint variables.

compared, because SICStus is implemented in C-code and can thus be expected to execute faster than a pure Java-program like **J.CP** .

In all benchmarks no search is performed, because propagation leads to a solution. The `order n`-benchmark can be described as follows $\forall x_i \in \{x_1, x_2, ..., x_\mathtt{n}\}$: $x_i \in \{1, 2, ..., \mathtt{n}\} \wedge x_{i-1} < x_i$, its implementation is resented in Figure 5. The only solution thus is $x_i = i$ for all $i \in \{1, 2, ..., \mathtt{n}\}$. In `reorder n`, we retract the con-

```
order(Solver cs, int N){
     FDVar[] vars = new FDVar[N]; // define FD-Variable array
     Lt[] lt = new Lt[X];   // define array for Lt-Constraints
     for(int i = 1; i< N; i++){
         vars[i] = new FDVar();   // create FD-constraint-variable
         (new In(vars[i],0,N-1)).post(cs);  //set 'vars[i] in 0..N-1'
         if(i!=1){
              lt[i] = new Lt(vars[i-1], vars[i]);  // create constraint
              lt[i].post(cs);   // post 'vars[i-1] #< vars[i]' in cs
     }   }
     // lt[N/2].retract();  // uncomment this for reorder(N)
     cs.awaitReliability(10);  // wait for completed propagation
}
```

Fig. 5. Example constraint program implementing the `order`-benchmark

straint $x_{n/2-1} < x_{n/2}$ after the execution of `order`, so that every x_i has a domain containing $n/2$ elements. In SICStus, retraction is simulated by backtracking to the named constraint and reposting all constraints, that were called after that. All shown results are average values of 10 tests.

	J.CP synchronous	**J.CP** asynchronous	SICStus/jasper
`order 2`	28	329	712
`order 100`	398	699	790
`order 200`	1183	1253	932
`order 500`	6232	5340	2075
`order 1000`	23194	14344	6812
`reorder 100`	446	697	814
`reorder 200`	1392	949	992
`reorder 500`	7752	2834	2181
`reorder 1000`	28213	7108	7374

Fig. 6. Comparison of **J.CP** with and without asynchronous constraint solving and SICStus v3.8.5 on a dual Pentium III Linux PC

The speedup of the asynchronous approach results from the differing order of execution of the propagation algorithms. With the execution of the constraint retraction before the propagation of all other constraints is finished we could even achieve a speedup of **J.CP** to jasper in the `reorder 1000` benchmark. This results from the higher priority of constraint retraction tasks. They are executed before all other constraints are finished, thus after its executeion only half of the propagation has to be performed.

But also in the `order` benchmark asynchronous execution is faster because in the average case less propagation is performed. Since the constraints are buffered, it is possible that due to the scheduling-algorithm a constraint c_1 that was posted or reposted after another constraint c_2 is executed first. We demonstrate the possible speedup achieved by this for `order 4` in Figure 7. On the left hand

normal	optimal
$x_1 \in 1..4, x_2 \in 1..4, x_3 \in 1..4, x_4 \in 1..4$	$x_1 \in 1..4, x_2 \in 1..4, x_3 \in 1..4, x_4 \in 1..4$
$x_1 < x_2$ (from buffer) $x_1 \in 1..3, x_2 \in 2..4, x_3 \in 1..4, x_4 \in 1..4$	$x_1 < x_2$ (from buffer) $x_1 \in 1..3, x_2 \in 2..4, x_3 \in 1..4, x_4 \in 1..4$
$x_2 < x_3$ (from buffer) $x_1 \in 1..3, x_2 \in 2..3, x_3 \in 3..4, x_4 \in 1..4$	$x_2 < x_3$ (from buffer) $x_1 \in 1..3, x_2 \in 2..3, x_3 \in 3..4, x_4 \in 1..4$
$x_1 < x_2$ (propagation) $x_1 \in 1..2, x_2 \in 2..3, x_3 \in 3..4, x_4 \in 1..4$	$x_3 < x_4$ (from buffer) $x_1 \in 1..3, x_2 \in 2..3, x_3 = 3, x_4 = 4$
$x_3 < x_4$ (from buffer) $x_1 \in 1..2, x_2 \in 2..3, x_3 = 3, x_4 = 4$	$x_2 < x_3$ (propagation) $x_1 \in 1..3, x_2 = 2, x_3 = 3, x_4 = 4$
$x_2 < x_3$ (propagation) $x_1 \in 1..2, x_2 = 2, x_3 = 3, x_4 = 4$	$x_1 < x_2$ (propagation) $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4$
$x_1 < x_2$ (propagation) $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4$	

Fig. 7. Comparison between optimal and normal execution order of the `order 4` problem. In "normal" execution constraint-repostings as a consequence of domain-restrictions, are executed immediately.

side of the table the normal execution order, that is used by SICStus is shown. Whenever a constraint restricts the domain of a known variable, all possible propagation steps are executed. This leads to the reduction of the domains with exactly one value. If this order is changed, like it is shown on the right hand side of the table, more extensive reductions may occur (last line). This yields less propagation in the solving process and is thus a potential speedup. This speedup grows together with the problem size in our benchmark, because the larger the restriction of a domain is, the more effect will the associated propagation have. Since all posted constraints in the benchmarks are instances of the same class, they are executed with the same priority. Thus the described synergy-effects occur only accidently.

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] Eric Bruchez and Marc Torrens. *Java Constraint Library.* `http://liawww.epfl.ch/~torrens/Project/JCL/`, 1996.

[4] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag, 1997.

[5] Andy Hon Wai Chun. Constraint programming in java with jsolver. In *Proc. Practical Applications of Constraint Logic Programming PACLP99*, 1999.

[6] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 1996.

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

[8] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[9] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL-87*, 1987.

[10] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[11] J.-F. Puget. A c++ implementation of clp. Technical report, ILOG, 1994. In Ilog Solver Collected papers.

[12] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[13] Makoto Yokoo and Edmund H. Durfee. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), 1998.