Constraint Processing with Reactive Agents

Georg Ringwelski* and **Richard J. Wallace**
Cork Constraint Computation Center†, UCC, Cork, Ireland
g.ringwelski|r.wallace@4c.ucc.ie

Abstract

The integration of methods of Constraint Programming and Multi-Agent-Systems is discussed in this paper. We describe different agent topologies for Constraint Satisfaction Problems and discuss their properties with a special focus on dynamic and distributed settings. This motivates our new architecture for constraint processing with reactive agents. The resulting systems are very flexible and can be used to process dynamic and distributed problems. We define the local behaviors of the agents in this new approach and verify their collective behavior in the context of CSP.

Introduction

In Multi-Agent-Systems (MAS), local behaviors for the agents are defined such that the global system will behave in an intended way. The methods for this implementation are rather restricted compared to traditional software systems. There is no global knowledge or shared memory that can be accessed by the agents and the only (very inefficient) way to communicate with other agents is by sending and receiving messages. The benefit of MAS on the other hand is that the resulting systems are much more flexible in many respects. Since agents only use little knowledge from its closest environment, they are much more tolerant against changes of the overall system or unexpected situations that may result from system faults or other agents that don't behave in the expected way. Due to the message-based communication, agents are widely used in distributed software environments. Especially in such settings, the flexibility of agents is important, since hardware is more likely to fail and intruders may try to violate the system.

The motivation of our research is to make use of the flexibility of MAS in distributed and dynamic constraint processing. We investigate different ways to process and solve Dynamic, Distributed Constraint Satisfaction Problems (DDCSP) with MAS. This yields a CP framework that is more flexible to use than current constraint solvers or algorithms. This paper describes the basis for these investigations. We motivate, describe and verify our new, purely agent-based architecture for DDCSP processing.

*This work was partially funded by the Embark initiative of the Irish Research Council for Science, Engineering and Technology under grant PD/2002/21.

†The Cork Constraint Computation Center is supported from Science Foundation Ireland under Grant 00/PI.1/C075.

Background

CP Preliminaries

Solving a CSP is finding a variable assignment into a domain of values $ass : V \rightarrow D$, that satisfies all constraints. Every variable is associated a finite set of allowed values, called its (variable) domain, by $dom : V \rightarrow \mathcal{P}(D)$. Every constraint c is posted over a tuple of variables $var(c) \in \{(v_1, v_2, ..., v_n) \mid v_i \in V\}$ and defines a set of allowed tuples of values for these variables. A constraint is said to be satisfied, if in its variables at least one of the allowed tuples can be composed from the current variable domains. These constraint semantics are usually given by relations in the addressed constraint system: $sem(c) \subseteq dom^+(var(c))$, where $dom^+ : V \times V... \times V \rightarrow \mathcal{P}(D) \times \mathcal{P}(D)... \times \mathcal{P}(D)$ is the point-wise extension of dom to tuples of variables. In most commercial constraint processing systems, the allowed tuples $sem(c)$ are not stored explicitly (i.e. as tuples), but as boolean functions in the Cartesian product of all variable domains (so-called propagators (Schulte & Carlson 2002; Ringwelski 2003) or local consistency operators (Debruyne *et al.* 2003)). In addition to the satisfaction check, these propagators remove values from variable domains that are proven not to be part of any solution. For a detailed description of this idea please refer to (Codognet & Diaz 1996). Propagators are interpreted (Apt 1998) by monotonic and inflationary Domain Reduction Functions (DRF) red in $\underset{v \in V}{\times} \mathcal{P}(D)$ that reduce the domains of variables $V_{out} \subseteq V$ if applicable, such that $\forall v \in V_{out} : red \downarrow_v (dom(v)) \subseteq dom(v)$ and $\forall v \in V \setminus V_{out} : red \downarrow_v (dom(v)) = dom(v)$. The propagator has to be implemented such that the domain reduction matches $sem(c)$. The semantics of a constraint program is in this setting the fixed point of the repeated application of all propagators. If the implementations of all propagators are correct, this yields the same variable domains as $\bigwedge_{c \in C} sem(c)$.

With these notions, we can define some tackled problem classes of the Constraint Programming framework.

Definition 1 (CSP,DistrCSP) *A* **C***onstraint* **S***atisfaction* **P***roblem* (C, V, dom) *in a constraint domain* D *is given by a finite set* C *of constraints, a finite set* V *of variables and a total mapping* $dom : V \rightarrow \mathcal{P}(D)$*. Such that* $\forall c \in C : var(c) \subseteq V \wedge sem(c) \subseteq dom^{+}(var(c))$*. A* **Distr***ibuted* **CSP** (C, V, dom, A, loc) *is a CSP* (C, V, dom) *with an additional set of agents A and a relation* $loc \subseteq (V \cup C) \times A$*.*

These definitions address static closed world settings, where constraints, variables and their initial domains are known in advance. Real-world problems often require a more dynamic setting, where solutions are to be adapted to new knowledge. This new knowledge can arise in our framework in the following ways: the addition of new constraints and thus their incremental integration into a (partially) processed CSP, and the exclusion of former beliefs and thus the retraction of constraints.

The addition or removal of variables from a CSP is not tackled in this paper. As in Logic Programming we assume a finite, but sufficiently large set of variables V to be given in advance and that for all constraints $var(c) \subseteq V$ holds. Consequently variable assignments are *partial* mappings and solutions are assignments that are defined for all variables occurring in any constraint. These assumptions do not restrict generality, because variables can always be constructed on demand.

Definition 2 (DynCSP,DDCSP) *A* **Dyn***amic* **CSP** *is a sequence* $a_o, a_1, ...$ *of CSPs, where* $a_0 = (V, C_0, dom_0)$ *with* $\forall v \in V : dom_0(v) = D$ *and* $a_n = (V, C_n, dom_n)$*, with* $\forall v \in V : dom_n(v) \subseteq D$ *and* $C_n = C_{n-1} \cup \{c\}$ *or* $C_n = C_{n-1} \setminus \{c\}$*. A* **D***ynamic* **D***istributed* **CSP** *is a similar sequence of Distributed CSPs* $a_0 = (V, \emptyset, dom_0, A_0, loc_0)$ *and* $a_n = (V, C_n, dom_n, A_n, loc_n)$*, where* $\forall a \in V \cup (C_n \cap C_{n-1}) : loc_n(a) = loc_{n-1}(a)$ *holds. The set of Agents* A_n *is always updated to the codomain of* loc_n

MAS Preliminaries

In this section we want to clarify our understanding of some basic notions of Multi-Agent-Systems. We want to make clear, what we mean as we have observed a lot of confusion in this area due to different understanding of basic vocabulary. We consider an agent a software object that has the following properties:

- It can communicate with other agents by messages
- It holds its own resources: memory and computation time
- It holds a partial representation of its environment, typically proxies[1] of other agents
- It is driven by a set of objectives it tries to achieve by its behavior.

For this, agents require some communication infrastructure (middleware), which we do not discuss in this paper. We make the general assumption, that every message occasionally arrives at its receiver. A MAS is then given by a set of agents and the middleware in an open computing system.

[1] A proxy is a "surrogate or placeholder of another object to controll access to it" (Gamma *et al.* 1995).

Regarding different research areas we separate two main approaches to MAS: Cognitive/intelligent agents and reactive agents (Ferber 1999). Cognitive agents, mainly known from Distributed AI, are 'intelligent' entities, that can solve parts of the global problem themselves and communicate their results with similar agents. Every cognitive agent makes use of its own knowledge base and has problem solving capabilities. The MAS then consists of a rather small number of such agents, typically one per physical machine.

In contrast to cognitive agents, reactive agents are not 'intelligent' themselves. They just provide some simple behaviors such that the overall MAS will be 'intelligent'. This is, they react on events without any internal planning, or examinations of the history or environment to solve complex problems cooperatively.

Download!

We have implemented a rudimentary demo tool for our new approach, which can be downloaded from `http://www.4c.ucc.ie/~gringwelski/racp`. It provides some Java classes and documentation for DDCSP processing in the Internet and a GUI to define problems and observe variable domains.

Identifying Agents for DDCSP (Related Work)

Given the definition of agents from the MAS preliminaries, different agent topologies can be found for DDCSP processing. Regarding the above enumeration of agent properties the first step towards agents is to identify separate objectives that can be achieved with local knowledge for every class of agents. In this section, we discuss such identifications for general DDCSP, which use both, the cognitive and the reactive agents approach.

Cognitive and Intelligent Agents

In distributed CSP applications today, the cognitive agents approach (Decker, Durfee, & Lesser 1989) is widely used. Every agent uses a local constraint reasoning engine and communicates its results with other agents. In this architecture, usually exactly one agent is running on every participating host, that processes a CSP in its local standard solver (e.g. (Schlenker 2003)). The main advantage of this architecture is, that the existing and efficient constraint-implementations of established solvers can be used for DistrCSP. Message passing is only used where it is necessary and not within one process, where method invocation is much faster. A DistrCSP (C, V, dom, A, loc) is in these settings partitioned (Hannebauer 2001) into a conjunction of one CSP per Agent: $\bigwedge_{a \in A} (C/_{\equiv}, V/_{\equiv}, dom \downarrow_{\{v \in V | loc(v) = a\}})$, where $\equiv = ker(loc)$ is the equivalence relation defined by the kernel of loc. Domain specific protocols for such systems are used to communicate the results by messages between separate agents. The objectives of such agents are determined by the application domain. In many cases, this will include consistency maintenance in its local CSP.

In the Mozart and the j.cp (Ringwelski 2002) solver, a generic approach to DistrCSP with cognitive agents was chosen: Two agents are connected

by variables, of which a copy exists in both agents. The protocol used in these systems implements an equality-constraint between these two copies of the semantically identical variable. A DistrCSP $(C, V \cup V', dom, \{A_1, A_2\}, loc(v) = \begin{cases} A_1, \text{ if } v \in V, \\ A_2, \text{ if } v \in V' \end{cases}$ for example is then internally extended to $C \cup \{eq_1, ..., eq_n\}, V \cup V' \cup \{v_1, v'_1, ..., v_n, v'_n\}, dom, \{A_1, A_2\}, loc,$ where $\forall i \in [1..n] : var(eq_i) = (v_i, v'_i) \wedge (v_i \in V \vee v'_i \in V') \wedge loc(v_i) = A_1 \wedge loc(v'_i) = A_2$ holds. The semantics of eq_i is equality, which propagates values (forward-checking) in the case of Mozart and all domain reductions (full look-ahead) in j.cp.

The difficulty with cognitive agent approaches to DistrCSP often arises with the implementation of search algorithms. Search algorithms, that make use of global knowledge cannot not be applied, because this is not accessible from the local view of an agent. Since most known algorithms require such knowledge (e.g. states in BT or BJ, evaluation functions in LS) some special effort has to be taken to apply them in distributed settings. This could be encapsulation of search spaces (Schulte, Smolka, & Würtz 1994), synchronization (Meisels & Razgon 2001) or using constraint retraction for erroneous instantiations (Ringwelski 2003). In special application domains, one can also consider (generally incomplete) specialized protocols that communicate just what is needed (Schlenker 2003; Hannebauer & Müller 2001) in the application domain.

For DynCSP, cognitive agents inherit the capabilities of the used local solvers. If dynamic solvers are used locally, a distributed network of such agents will support DDCSP. The local solvers are used as a local interface to the distributed problem. Currently, however, only j.cp as a generic solver for DDCSP is (informally) available. While incremental constraint addition is supported in many solvers (e.g. CLP), constraint retraction has been a crucial research topic for more than 5 years, but has not yet found its way into any popular CP system. The use of most existing retraction algorithms are almost excluded from DistrCSP, because they make intensive use of global knowledge ((Wolf 2001; Debruyne *et al.* 2003; Bessiére 1991) and others). The ACS retraction algorithm (Ringwelski 2003), which decentralizes such structural information can be used for DDCSP processing. However, with this algorithm a great deal of communication is required to supply the necessary information to supply the necessary information to all agents that need it.

Reactive Agents

Reactive agents are defined to be small, non-'intelligent' entities that cooperatively behave 'intelligently' (Ferber 1999). In the context of CSP this would mean, a reactive agent has no capabilities in constraint solving, but represents one essential object that behaves in a way that will make the overall system solve CSPs. The main advantage of reactive agents is their flexibility. If the behaviors of the agents are defined properly, it is expected, that the overall system will always behave in the intended way. Thus, if an agent dies or behaves in an unexpected way, the overall system will still be able to converge towards (if not actually reach) the overall solution. This topic of fault-tolerance will be a central point of our future research with our new CP platform.

The most popular approach to DistrCSP uses such agents: In complete Asynchronous Backtracking search algorithms (Yokoo & Hirayama 2000) every variable is represented by one agent. The (binary) constraints are stored explicitly (i.e. as tuples of allowed values) in the variable agents in this family of algorithms (Yokoo & Hirayama 2000; Silaghi, Sam-Haroud, & Faltings 2000; 2001). In a DistrCSP (C, V, dom, A, loc), the relation loc will thus map a set of agents to every constraint and be a bijective function on V. The objective of the variable agents is to find a value that satisfies all constraints. This is implemented by storing and communicating nogoods, which are provably unsuccessful partial variable assignments. Unfortunately, a global structure between agents must be imposed in these algorithms, which makes them intolerant to system faults. The reason for this is, that the agents are not designed consequently as reactive agents: They store some history of their environment (namely the nogoods) which seems already to be too much external knowledge to maintain the flexibility of reactive agents. If one variable-agent stops behaving in its usual way, these algorithms will not be able to compute anything further. Maybe, this is the price to pay for complete search. Being a terminating, non-incremental algorithm, ABT (and variants) cannot be used directly for truth-maintenance in a dynamic CSP. However, an agent that imposes an ABT search within a DDCSP system would be a natural integration of such an algorithm into a dynamic constraint processing system. But what kind of an agent should be used to represent and invoke the search?

This leads to another kind of reactive agents in CSP, which we found most reasonable in our current research: Constraints as reactive agents. The mentioned search algorithm would then represent a labeling-"constraint" as it is provided by many incremental CP systems. Much more intuitive are, however, constraint agents that represent regular constraints, such as arithmetic (in-)equations or global constraints for example. The main objective of such a reactive constraint-agent is to reduce variable domains in accordance to its intended semantics. The behavior that implements this objective is a regular propagator which is triggered, whenever it might infer new domain reductions.

When identifying constraints as agents, the storage of constraints in variables should be omitted in order not to store same things multiple times. Reactive agents that represent variables will have to take care that propagators are invoked, whenever they could detect an inconsistency or propagate. For this, the variable agent will have to send its constraints a message, if its domain was reduced. In our current research, we identified this the main objective of variable agents.

Conception of RACP Systems

In order to integrate the advantages of both, the cognitive and the reactive approaches we define RACP systems for DDCSP processing in two layers of agents (Fig. 1):

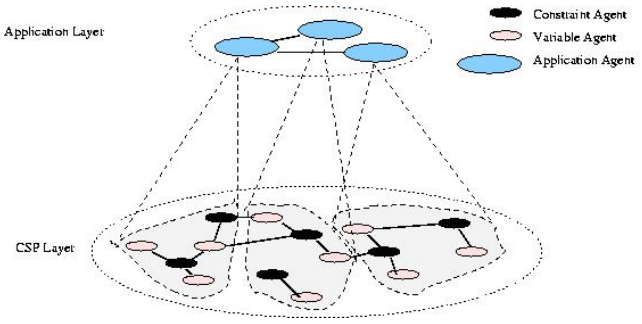

Figure 1: The application layer and the CSP layer.

1. The application layer, which contains cognitive agents and user interfaces
2. The CSP layer, which contains two sorts of reactive agents: Variables and constraints

The agents (AA) in the application layer can take care that communication between agents that are started in its own process communicate by method invocation to achieve the performance of monolithic solvers for the local problems. The agents in the CSP layer (variable agents VA and constraint agents CA) communicate with other agents in a way that is established by the creating AA. This will allow to preserve most advantages of cognitive agent approaches or even centralized solvers in RACP. On the other hand, the VA and CA perform the DDCSP processing on themselves and do not rely on any global structure or the existence of other agents. As we will show in the next section, they use local knowledge exclusively to cooperatively process the DDCSP which they make up themselves. This will preserve the main advantage of reactive agents: Flexibility and fault-tolerance.

The Local Behaviors in RACP

In this section we enumerate the behaviors that VA and CA implement in order to achieve their objectives as described above. In the following section we will evaluate these behaviors with respect to the overall behavior of the resulting Multi-Agent-System.

Variables

(1) Upon its construction, the VA stores its finite initial variable domain `dom` locally.

(2) Whenever a `confirm(n)` message arrives, the proxy of its sender c (a CA) will be stored in the set of adjacent constraints `con`[2]. Furthermore the proxies of variables in `n` are added to the locally stored set `neighbors` and a `propagate_v(dom)` message is sent to c.

(3) Whenever a `propagate_c(d)` message arrives, it assigns `dom := dom` $\cap$ `d` and sends a `propagate_v(dom)` message to all $c \in$ `con`, that might infer new domain reductions from that.

(4) Whenever a `disconfirm` message arrives, its sender `c` (a constraint proxy) is removed from `con`, `dom` is relaxed to its initial size, `c` is stored in the set `relaxedFor` in `dom` and all $v \in$`neighbors` are sent a `relax(c)` message. After that, a `propagate_d(dom)` message is sent to all $c \in$ `con`.

(5) Whenever a `relax(c)` message arrives and it has not relaxed for `c` before, it relaxes `dom` to its initial domain, stores `c` in the set `relaxedFor` in `dom` and sends all $v \in$ `neighbors` a `relax(c)` message.

Constraints

(6) Upon its construction it stores the proxies of all its variables in V_c and sends them a `confirm(`V_c`)` message.

(7) Upon receiving `retract` method, it sends all $v \in V_c$ a `disconfirm` message.

(8) Whenever it receives a `propagate_v(d)` message from any $v \in V_c$, it stores `d` in its local copy of the variable domains `dom[`v`]` and calls its propagator method, iff

(8a) A copy of the domain of all $v \in V_c$ was stored before.

(8b) In `dom[`v`]` for all $v \in V_c$, the sets `relaxedFor` are identical.

The propagator will then infer new variable domains $\mathtt{d}_v$ for its variables from its current knowledge `dom` and send a `propagate_c(`$\mathtt{d}_v$`)` message to all $v \in V_c$ for which $\mathtt{d}_v \subset$`dom[`v`]` holds.

Search

As we have indicated before, search can be integrated as a constraint in RACP. The declarative semantics of such a constraint is: "The domain of all my variables has a cardinality of one". Established distributed search algorithm can be integrated in RACP by encapsulating them in such global constraints. If synchronization is required (e.g. in (Meisels & Razgon 2001)), distributed snapshots (Mattern 1993) can be used to make use of the system inherent propagation. If this is not intended, as in ABT for example, the search has to be performed within the components created for this "constraint" before its result is used in the DDCSP.

However, such global "labeling constraints" are not really compatible with the idea of reactive agents, because they use global knowledge such as snapshots for example. Our research thus focuses on de-centralized search algorithms. Using variable instantiation constraints, propagation and retraction, we define local behaviors directly related to variables that try to make the overall system converge to a solution. A first prototype of a unary labeling constraint is already implemented for RACP. If a value for a variable is searched for, such a constraint is posted over it. If a global solution is searched for, such a constraint is posted over every variable. The propagator of this constraint reacts on the size of the domain of its variable in the following way: If the domain has more than one value, it chooses one and posts an instantiation for it; if the domain has exactly one value it

[2]Actually, constraints are stored in separate sets. In a similar way to (Codognet & Diaz 1996) one set is stored for specified properties of the variable domain. Thus only the constraints that might infer further domain reductions are triggered, which improves the overall performance

does nothing because it is satisfied; if the domain is empty and an instantiation was posted before, it retracts the instantiation. In the latter case it will be triggered as soon as the retraction is finished and has then the chance to find another value for the instantiation. If the domain is empty, but no instantiation was posted before, another source of the inconsistency has to be found and the propagator does nothing. A set of such unary labeling constraints yields an incomplete search algorithm.

The termination of such de-centralized search algorithms will be hard to prove and completeness will never be achieved (Collin, Dechter, & Katz 1999). The latter is obvious, since we use local search algorithms, however, in many real-world problems solutions can be found with this class of algorithms. Termination in dynamic and continious systems is a crucial topic. We think, in our targeted application domains, a globally stable state will be impractical to wait for, since many users in a large network can always change the problem. Every client can be expected to be interested only in a small part of the CSP, for which it can read the current knowledge. Due to the complexity of the system, this can already be outdated, but it is the best, that is currently available.

The Global Behavior of RACP

RACP can be used for processing any DDCSP with $a_n = (V, C_n, dom_n, A_n, loc_n)$. The functions loc_n and the sets of agents A_n are a priori given as the CSP layer defined by application agents. This topology cannot change (except through constraint addition or removal) during DDCSP processing as we do not support mobile agents (see Def. 2).

We verify the correctness of the possible transformations (adding and removing constraints) in DDCSP processing by induction over the achieved global states. We thus prove that correct variable domains with respect to all currently valid constraints are maintained. We don't prove that the system will find a solution or that a search algorithm terminates. Correct means in this context, that dom_{n+1} is the fixed point of Chaotic Iteration (Apt 1998) of the DRF that are implied by the propagators of all $c \in C_{n+1}$. Due to lack of space we only give the main outline of the proof.

The operations are preformed incrementally and we start from the empty set of constraints in $a_0 = (V, \emptyset, dom_0, A_0, loc_0)$. As the anchor of induction we use this trivially correct state. From there, we assume that dom_n is correct and show for both transitions, that they infer correct domains dom_{n+1}.

Adding Constraints

Let's assume, that a constraint c that is implemented by a propagator method p_c, which implies a DRF f_c is added to a_n then we get a globally stable state a_{n+1} if p_c terminates and creates finitely many constraints. This follows from the fact, that `propagate` messages are only sent upon domain reductions (3) and (8) and that we assume finite variable domains (1). Consequently, there can only be finitely many `propagate` messages and thus also invocations of constraint propagators (8).

If $c \in C_n$, then dom_n was computed including p_c and no more domain reductions can be inferred from (3) or (8), such that $dom_{n+1} = dom_n$. In this case $C_n = C_{n+1}$ hold and thus dom_n represents the correct variable domains in a_{n+1}.

If $c \notin C_n$, we get dom'_{n+1} in a_{n+1}. This is sound wrt. dom_{n+1}, because dom_{n+1} is computed with f_c. In other words, no un-intended values are excluded from the variable domains, because the semantics of a constraint *is* its implied DRF. In order to show, that dom'_{n+1} is complete, we make a proof by contradiction: Assuming, dom'_{n+1} is not complete and thus $\exists v \in V : dom_{n+1}(v) \subset dom'_{n+1}(v)$. From this and the uniqueness of dom_{n+1} (Apt 1998) follows, that $\exists c \in C_{n+1}, v \in V : f_c(dom_{n+1}(V)) \downarrow_v \subset f_c(dom'_{n+1}(V)) \downarrow_v$ and this would mean, that a propagator p_c that might infer further domain reductions was not executed in RACP. But since we assume a running communication infrastructure, this is a contradiction to (3) and (8), because of the correct integration (6)(2) of every constraint in the DDCSP.

Deleting Constraints

The constraint retraction algorithm in this approach is rather trivial: Enlarge all possibly affected variable domains to their initial size and then propagate all their constraints. This algorithm is not "as incremental as" those mentioned above, but we found it the most appropriate for the use in distributed settings. The main reason for this is, that it does not require much synchronization or centralization. This algorithm can be executed in parallel and doesn't have to wait until the complete CSP is relaxed. According to (8b), every constraint can continue pursuing its main objective (domain reduction) as soon, as its closest environment , i.e. *its* variables, has finished the relaxation. Constraint retraction in RACP does not require any further synchronization. In general, constraint retraction does not affect found solutions, since every solution will remain valid in a relaxed problem. We have observed in previous research on constraint retraction algorithms, that this trivial algorithm is in many real-world cases (non-binary, one-component CSPs) similarly efficient as more sophisticated retraction algorithms (e.g. (Georget, Codognet, & Rossi 1999)). The extra cost for propagation is more than made up in distributed settings by the fact, that no structural information on multiple agents, such as justifications (Debruyne *et al.* 2003) for example, must be considered for the retraction algorithm.

Now we sketch the proof of correctness of constraint retraction in RACP, we still assume in our induction that dom_n is correct for any a_n during DDCSP processing. The constraint retraction algorithm described by (7),(4) and (5) will terminate, since variables only relax their domains (5), if they did not so before. This imposes a wave of markers (Mattern 1993), which will always terminate, as we assume finite problems (Def. 1). The following propagation (4) will also terminate as shown in the previous section. The resulting globally stable state with dom'_{n+1} is correct wrt. the result dom_{n+1} of Chaotic Iteration of the DRF implied by C_{n+1}. This follows from the fact, that all domains are relaxed and that the sound and complete propagation is reinvoked after the relaxation (4).

The main difficulty in these two steps is, not to confuse

old and new knowledge in the process of propagation. As known from the theory of Chaotic Iteration, domain enlargements must not be performed during propagation in order to reach a well defined fixed point. However, the local relaxation enlarges variable domains in (4) and (5). We solve this problem by separating the new propagation phase from the old one by a global cut (Mattern 1993) which is imposed by the sets `relaxedFor` in the domains. With this storing of the retracted constraints a variable has already relaxed for in (4) we omit this source of confusion. Restricting constraint propagation (8b) to the cases, where all used domains are from "after the cut", i.e. their variables have fished the relaxation, synchronizes the relaxation with the new propagation in every constraint locally. Overall we thus impose a new propagation after every relaxation. The constraints will not contribute to this new propagation process, before all its variables have finished their relaxation phase, i.e. before exclusively new knowledge is used.

Conclusion and Future Work

We have discussed different ways to integrate Multi-Agent-Systems and Constraint Programming. This motivated the new RACP architecture for DDCSP processing in two layers of agents. In the application layer, cognitive agents are used to make up an open system of application programs which create variables and constraints on the CSP layer. Every constraint and variable is a reactive agent on the CSP layer.They cooperatively solve distributed CSPs dynamically and preserve the potential of fault-tolerance and flexibility that is achieved with reactive agents. The new RACP framework was defined and verified.

In order to find solutions in DDCSP, search algorithms have to be used. We described very briefly a first decentralized algorithm, which is to find global solutions without using any global control. One main branch of our future research will concentrate on such algorithms. This will yield fault-tolerant and flexible agents that (re-)compute solutions for changing CSPs with local behaviors of the reactive agents.

The general topic of our future research will be to investigate fault-tolerance for distributed CSP. We plan to define a semantics, that takes care of agents that do not behave in the expected way, of messages that don't arrive and knowledge, that disappears. With this, we plan to find ways, to omit the currently in DistrCSP generally made but unrealistic precondition of an error-free communication infrastructure.

References

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

Bessiére, C. 1991. Arc-consistency in dynamic constraint satisfaction problems. In *Proc. AAAI'91*, 221–226.

Codognet, P., and Diaz, D. 1996. Compiling constraints in clp(fd). *Journal of Logic Programming*.

Collin, Z.; Dechter, R.; and Katz, S. 1999. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*.

Debruyne, R.; Ferrand, G.; Jussien, N.; Lesaint, W.; Ouis, S.; and Tessier, A. 2003. Correctness of constraint retraction algorithms. In *Special Track on Constraint Solving and Programming of FLAIRS 2003*.

Decker, K.; Durfee, E.; and Lesser, V. 1989. Evaluating research in cooperative distributed problem solving. *Distributed Artificial Intelligence* 2:485–519.

Ferber, J. 1999. *Multi-Agent-Systems, An Introduction to Distributed Artificial Intelligence*. Addison-Wesley.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns*. Reading, MA: Addison Wesley.

Georget, Y.; Codognet, P.; and Rossi, F. 1999. Constraint retraction in CLP(FD). *Constraints* 4(1):5–42.

Hannebauer, M., and Müller, S. 2001. Distributed constraint optimization for medical appointment scheduling. In *Proceedings of the Fifth International Conference on Autonomous Agents (AGENTS-2001)*.

Hannebauer, M. 2001. *Autonomous Dynamic Reconfiguration in Collaborative Problem Solving*. Ph.D. Dissertation, TU Berlin. also published as Springer LNCS 2427 in 2002.

Mattern, F. 1993. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing* 18(4):423–434.

Meisels, A., and Razgon, I. 2001. Distributed forward checking with dynamic ordering. In *Proc. CP01 workshop on collective search algorithms*.

Ringwelski, G. 2002. Object-oriented constraint programming with j.cp. In Coello, C., ed., *Advances in Artificial Intelligence, MICAI2002*, LNAI 2313, 194–203. Springer.

Ringwelski, G. 2003. *Asynchrones Constraintlösen*. Ph.D. Dissertation, Technical University Berlin. `edocs.tu-berlin.de/diss/2003/ringwelski_georg.htm`.

Schlenker, H. 2003. Distributed constraint-based railway simulation. Doctoral Programme of CP03.

Schulte, C., and Carlson, M. 2002. Finite domain constraint programming systems. Tutorial at CP02 conference, Ithaca, USA. `www.it.kth.se/schulte/talks/FD`.

Schulte, C.; Smolka, G.; and Würtz, J. 1994. Encapsulated search and constraint programming in Oz. In Borning, A., ed., *Second Workshop on Principles and Practice of Constraint Programming*, LNCS 874, 134–150.

Silaghi, M.-C.; Sam-Haroud, D.; and Faltings, B. 2000. Asynchronous search with aggregations. In *Proc. AAAI/IAAI 2000*, 917–922.

Silaghi, M.-C.; Sam-Haroud, D.; and Faltings, B. 2001. Consistency maintainance for abt. In Walsh, T., ed., *Principles and Practice of Constraint Programming - CP 2001*, 271–285. Springer LNCS 2239.

Verfaillie, G., and Schiex, T. 1994. Solution reuse in dynamic constraint satisfaction problems. In *National Conference on Artificial Intelligence*, 307–312.

Wolf, A. 2001. Adaptive constraint handling with chr in java. In Walsh, T., ed., *Proc. CP2001*, LNCS 2239, 256–270. Springer.

Yokoo, M., and Hirayama, K. 2000. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems* 3(2):185–207.