Distributed Configuration as Distributed Dynamic Constraint Satisfaction

Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker

Institut für Wirtschaftsinformatik und Anwendungssysteme, Produktionsinformatik, Universitätsstrasse 65-67, A-9020 Klagenfurt, Austria,
{felfernig,friedrich,jannach,zanker}@ifi.uni-klu.ac.at.

Abstract. Dynamic constraint satisfaction problem (DCSP) solving is one of the most important methods for solving various kinds of synthesis tasks, such as configuration. Todays configurators are standalone systems not supporting distributed configuration problem solving functionality. However, supply chain integration of configurable products requires the integration of configuration systems of different manufacturers, which jointly offer product solutions to their customers. As a consequence, we need problem solving methods that enable the computation of such configurations by several distributed configuration agents. Therefore, one possibility is the extension of the configuration problem from a dynamic constraint satisfaction representation to *distributed* dynamic constraint satisfaction (DDCSP). In this paper we will contribute to this challenge by formalizing the DDCSP and by presenting a complete and sound algorithm for solving distributed dynamic constraint satisfaction problems. This algorithm is based on asynchronous backtracking and enables strategies for exploiting conflicting requirements and design assumptions (i.e. learning additional constraints during search). The exploitation of these additional constraints is of particular interest for configuration because the generation and the exchange of conflicting design assumptions based on *nogoods* can be easily integrated in existing configuration systems.[1]

1 Introduction

Dynamic constraint satisfaction is a representation formalism suitable for representing and solving synthesis tasks, such as configuration [11], [13]. These tasks have a dynamic nature in the sense that the set of problem variables changes depending on the initial requirements and the decisions taken during the problem solving process resulting in a reduction of the search space to relevant problem variables. In this context the notion of activity constraints discussed in [11] is well suited for stating decision criteria on the activity state of problem variables.

There is an increasing demand for applications providing solutions for configuration tasks in various domains (e.g. telecommunications industry, computer industry, or automotive industry). This demand is boosted by the mass customization paradigm and e-business applications. Especially the integration of

[1] This work was partly funded by the EU commission under contract IST-1999-10688.

L. Monostori, J. Váncza, and M. Ali (Eds.): IEA/AIE 2001, LNAI 2070, pp. 434–444, 2001.

configurators in order to support cooperative configuration such as supply chain integration of customizable products is an open research issue. Current configurator approaches [3] , [7] are designed for solving local configuration problems, but there is still no problem solving method which considers variable activity state in the distributed case. Security and privacy concerns make it impossible to centralize problem solving in one centralized configurator.

As a consequence, we have to extend dynamic constraint satisfaction to *distributed* dynamic constraint satisfaction. Based on asynchronous backtracking [14] we propose an extension for standard distributed CSP formalisms by incorporating activity constraints in order to reason about the activity states of problem variables. In the sense of distributed constraint satisfaction we characterize a Distributed Dynamic Constraint Satisfaction Problem (DDCSP) as a cooperative distributed problem solving task. Activity constraints, compatibility constraints as well as variables are distributed among the problem solving agents. The goal is to find a variable instantiation which fulfills all local as well as inter-agent constraints. In this context each agent has a restricted view on the knowledge of another agent, but does not have complete access to the remote agents knowledge base.

Asynchronous backtracking offers the basis for bounded learning strategies which supports the reduction of search efforts. This is of particular interest for integrating configurators. Configurators send configuration requests to their solution providing partners. These partners eventually discover conflicting requirements (i.e. *nogoods*) which are communicated back to the requesting configurator thus supporting the efficient revision of requirements or design decisions. Based on asynchronous backtracking we provide a sound and complete algorithm for distributed dynamic constraint satisfaction where a very limited *nogood* recording is sufficient.

In the following we give a formal definition for a DDCSP *(Section 2)* and show how to employ this formalism for representing a distributed configuration problem. Following this formal definition we propose an algorithm for solving DDCSP *(Section 3)*. We analyze runtime and space complexity of this algorithm and show completeness and soundness. Finally we discuss related work followed by general conclusions.

2 Distributed Dynamic Constraint Satisfaction Task

Before giving a definition of a DDCSP we recall the definition of a distributed CSP. A set of agents must find a solution for a distributed set of finite domain variables. There exist n agents, where each agent a_i has m variables x_{ij}, $i \in \{1..n\}$, $j \in \{1..m\}$. Agent variables have a priority denoted as p_{ij}. Each variable x_{ij} is assigned to a domain $D_l \in Doms = \{D_1, D_2, ...\}$, where $Dom\,(\mathrm{x}_{ij})$ denotes the domain of the variable x_{ij}. Constraints are distributed among agents, where C_{ik} denotes constraint k of agent a_i. A distributed CSP is solved *iff* $\forall \mathrm{x}_{ij}$, $\forall \mathrm{C}_{ik}$: x_{ij} is instantiated with one $d \in Dom\,(\mathrm{x}_{ij})$, and C_{ik} is true under the assignment $\mathrm{x}_{ij}{=}d$.

In order to formulate constraints on the activity state of problem variables, [11] propose four major types of activity constraints as follows. Note, that $x_{ij} \notin \{x_{pq} \wedge x_{vw} \wedge \ldots\}$ must hold, where x_{ij}, x_{pq}, and x_{vw} are different agent variables.

1. *Require Variable* ($\overset{RV}{\Rightarrow}$): the activity state of a variable depends on the value assignment to a set of active variables, i.e. $P(x_{pq},x_{vw},\ldots) \overset{RV}{\Rightarrow} x_{ij}$. P denotes a predicate determining whether the variable x_{ij} must be active or not.
2. *Always Require* ($\overset{ARV}{\Rightarrow}$): the activity state of a variable depends on the activity state of a set of other variables, i.e. $(x_{pq} \wedge x_{vw} \wedge \ldots) \overset{ARV}{\Rightarrow} x_{ij}$.
3. *Require Not* ($\overset{RN}{\Rightarrow}$): a variable must not be active if a certain assignment of a set of variables is given, i.e. $P(x_{pq},x_{vw},\ldots) \overset{RN}{\Rightarrow} x_{ij}$.
4. *Always Require Not* ($\overset{ARN}{\Rightarrow}$): a variable must not be active if a set of variables is active, i.e. $(x_{pq} \wedge x_{vw} \wedge \ldots) \overset{ARN}{\Rightarrow} x_{ij}$.

Based on the above definition of a distributed CSP and the notion of activity constraints we give a formal definition of a DDCSP. In order to determine whether a variable x_{ij} is active or not we associate a state variable with each x_{ij} denoted as $x_{ijstatus}$, where $Dom(x_{ijstatus}) = \{active, inactive\}$. Additionally there are two different types of constraints, namely *compatibility constraints* (C_C), which restrict the compatibility of variable assignments, and *activity constraints* (C_A), which constrain the activity state of constraint variables.

Definition 1 *Distributed Dynamic CSP (DDCSP)*
Given:

- *n agents, where agent* $a_i \in \{a_1, a_2, \ldots, a_n\}$.
- *Each agent* a_i *knows a set of variables* $V_i = \{x_{i1}, x_{i2}, \ldots, x_{im}\}$, *where* $V_i \neq \emptyset$ *and each* x_{ij} *has a dedicated state variable denoted as* $x_{ijstatus}$[2].
- *Furthermore,* $V_{istart} \subseteq V_i$ *denotes the initial active variables of agent* a_i, *where* $V_{start} = \bigcup V_{istart}$ *and* $V_{start} \neq \emptyset$. *Additionally, the following condition must hold:* $\forall x_{ij}: (x_{ij} \in \{V_{istart}\}) \Rightarrow x_{ijstatus} = active$, *i.e.* $true \overset{ARV}{\Rightarrow} x_{ij}$.
- *A set of domains* $Doms = \{D_1, D_2, \ldots\}$, *where each variable* x_{ij} *is assigned to a domain* $D_l \in Doms$ *and* $Dom(x_{ijstatus}) = \{active, inactive\}$.
- *A set of constraints* C *distributed among agents, where* $C = C_C \cup C_A$ *and* C_{Cik} (C_{Aik}) *denotes compatibility (activity) constraint* k *of agent* a_i.

Find:
A solution Θ *representing an assignment to variables which meets the following criteria:*

1. *The variables and their assignments in* Θ *satisfy each constraint* $\in C$, *i.e.* Θ *is consistent with* $C_C \cup C_A$.
2. *All variables* $x_{ij} \in V_{start}$ *are active and instantiated.*
3. *There is no assignment* Θ' *satisfying 1. and 2., s.t.* $\Theta' \subset \Theta$.

[2] In the following we denote x_{ij} as content variable, $x_{ijstatus}$ as status variable.

Following this definition we give an example of representing a distributed configuration task as DDCSP *(Figure 1)*. A car manufacturer (a_1), a chassis and motor supplier (a_2), and an electric equipment supplier (a_3) cooperatively solve a distributed configuration task. Shared knowledge is represented through variables belonging to common constraints. User requirements are provided as additional constraints, which are denoted as C_R= {(package=standard),(car-body=4door-limo)}, furthermore V_{1start} = {car-body, package}; V_{2start} = {transmission, motorization}; V_{3start} = {battery}. A solution for this configuration task is the following: {car-body= 4door-limo, package= standard, transmission= manual, motorization= 55bhp, battery= medium}. Note that the variables airbag, front-fog-lights, and electric-windows are not part of the above solution.

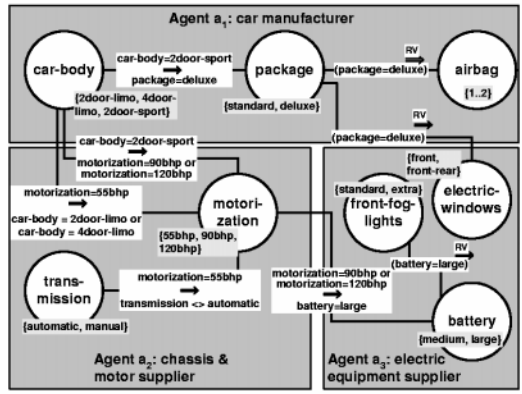

Fig. 1. Example: Distributed Configuration

3 Solving Distributed Dynamic Constraint Satisfaction Problems

In the following we discuss our extensions to *asynchronous backtracking* [14], which is a complete, asynchronous version of a standard backtracking algorithm for solving distributed CSPs. Problem variables are distributed among problem solving agents, where each agent a_i has exactly two variables (x_i and $x_{istatus}$)[3]. Each agent variable x_i has a unique priority p_i, furthermore x_i and $x_{istatus}$ always have the same priority[4]. We assume that constraints are binary, which is not limitating, because it is well known that any non-binary discrete CSP can be translated to an equivalent representation with binary constraints. For this

[3] The case of solving a distributed CSP with multiple local variables is discussed in [15]. In this case multiple virtual local agents (each working on a local variable) try to find a solution which satisfies all local constraints. The principles of asynchronous backtracking [14] remain the same for the case of multiple local variables.

[4] The lowest value represents the highest priority.

conversion two general methods are known: the dual graph method [6] and the hidden variable method [12]. We denote two agents whose variables participate in the same binary constraint as *connected*. Furthermore we can say that these links between variables are *directed*, because of the unique priority that is associated with each variable. The agent with the higher priority is sending the variable value (via *ok?* messages) after each change of his variable instantiation to the connected agents. The latter evaluate their local constraints and inform corresponding agents about local inconsistencies (via *nogood* messages). *Nogoods* represent conflicting variable instantiations which are calculated by applying resolution. Their generation is a potential source of inefficiency because of space complexity[5]. Variable assignments of value sending agents are stored as tuple *(j,(y_j,d))* in the local *agent_view* of the connected constraint evaluating agent, where y_j can either represent a content variable or a state variable.

3.1 Asynchronous Backtracking for DDCSP

Based on asynchronous backtracking we propose an algorithm for solving DDCSPs (*Algorithm 1*). The messages exchanged between agents have the following signatures, where j denotes the index of the message sending agent a_j.

- *ok?(j,(y_j,d)):* y_j denotes the variable name, and d the actual instantiation of y_j *(Algorithm 1 (a)).*
- *nogood(j,nogood$_j$):* *nogood$_j$* is a set of inconsistent variable instantiations represented as {*(k,(y_k,d)),...*}, where k denotes the index of the agent a_k storing variable y_k with value d *(Algorithm 1 (b)).*

Both, *ok*?, and *nogood* messages can contain content information as well as state information about communicated variables, e.g. *ok?(1,($x_{1status}$,inactive))* communicates activity information about variable x_1, *nogood(3,{(1, ($x_{1status}$,active)), (2,(x_2, 3))})* denotes the fact that either $x_{1status}$ must not be active or x_2 must not be instantiated with 3. If a tuple *(i,(x_i,d))* (x_i is a content variable) is added to the *agent_view*, an additional tuple *(i,($x_{istatus}$,active))* is added. If a tuple *(i,($x_{istatus}$,inactive)* is added to the agent_view, the tuple *(i,(x_i,d))* is deleted from the *agent_view.*

When the algorithm starts, all agents instantiate their active variables x_i, where $x_i \in V_{start}$, propagate their instantiations to connected agents and wait for messages. All other agents a_j propagate $x_{jstatus}$=*inactive* to connected agents[6]. When an agent receives a message, it checks the consistency of its local *agent_view (Algorithm 1 (c)).* Similar to the ATMS-based approach discussed in [11] our algorithm contains *two* problem solving levels.

First, all local activity constraints are checked in order to determine the *Activity State Consistency.*

[5] In *Section 3.3* we show how space complexity can be significantly reduced.
[6] This initialization is not included in *Algorithm 1.*

Definition 2 *Activity State Consistency*
Activity State Consistency is given, iff $x_{selfstatus}$ is consistent with agent_view, i.e. all evaluated activity constraints are true under the value assignments of agent_view, and all communicated nogoods are incompatible with agent_view[7].

The variable $x_{selfstatus}$ represents the state variable of the local agent a_{self}[8]. The function *ASC (c.1)* checks the *Activity State Consistency*, tries to instantiate $x_{selfstatus}$ with a consistent value if needed, and returns *true* if *Activity State Consistency* is given, otherwise it returns *false*. If an inconsistent *Activity State* is detected, e.g. there is a contradiction between activity constraints, and $x_{selfstatus}$ can not be adapted consistently, *nogoods* are calculated including the domain constraints of $x_{selfstatus}$ and backtracking is done *(c.7)*. If *Activity State Consistency* is given and $x_{selfstatus}$ has been changed to inactive, an *ok?* message containing the new variable state is sent to the constraint evaluating agents *(c.6)*.

Second, if *Activity State Consistency* is given *(c.1)* and the local variable is active *(c.2)*, the algorithm checks the *Agent Consistency*.

Definition 3 *Agent Consistency*
Agent Consistency is given, iff the agent is in a Consistent Activity State and x_{self} is consistent with agent_view, i.e. all evaluated compatibility constraints are true under the value assignments of agent_view, and all communicated nogoods are incompatible with agent_view.

The variable x_{self} represents the content variable of the local agent a_{self}. The function *AC (c.3)* checks the *Agent Consistency*, tries to instantiate x_{self} with a consistent value if needed, and returns *true* if *Agent Consistency* given, otherwise it returns *false*. If no *Agent Consistency* is given and x_{self} can not be instantiated consistently, *nogoods* are calculated including the domain constraints of x_{self} and $x_{selfstatus}$, and backtracking is done *(c.5)*. Else, if the value of x_{self} has been changed, an *ok?* message is sent to the connected constraint evaluating agents of a_{self} *(c.4)*.

Algorithm 1 *Asynchronous Backtracking for DDCSP*[9]

```
(a) when received (ok?(j,(yj,d))) do
      add {(j,(yj,d))} to agent_view;
      xselfstatusold ← xselfstatus; xselfold ← xself;
      check_agent_view; end do;
(b) when received (nogood(j,nogoodj)) do
      add nogoodj to nogood_list[10];
      if ∃ (k,(xk,d)) in nogoodj:
                  xk ¬connected[11] then
```

[7] *Agent_view* is compatible with a *nogood*, *iff* all *nogood* variables have the same value as in *agent_view*, $x_{selfstatus}$, and x_{self}.

[8] *self* denotes the index of the local agent a_{self}.

[9] The algorithm does not include a stable-state detection. In order to solve this task, stable state detection algorithms like distributed snapshots [2] are needed.

[10] The *nogood_list* contains the locally stored *nogoods*.

[11] In order to check the *nogood*, all variables of connected agents part of the *nogood* must be represented in the *agent_view*.

```
        request a_k to add a link to self;
        add current {(k,(x_k,d))} to agent_view;
      end if;
      x_selfstatusold ← x_selfstatus; x_selfold ← x_self;
      check_agent_view;
      if x_selfold = x_self ∧
            x_selfstatusold = x_selfstatus then
        send (ok?,(self,(x_self,d_self))) to a_j;
      end if; end do;
(c) procedure check_agent_view;
(c.1) if ASC(agent_view, x_selfstatus) then
(c.2)   if x_selfstatus = active then
(c.3)     if AC(agent_view, x_self) then
            if x_self ≠ x_selfold ∨
                  x_selfstatus ≠ x_selfstatusold then
(c.4)         send (ok?(self,(x_self,d_self)))
              to connected constraint evaluating agents;
            end if;
(c.5)     else nogoods←{K_s|K_s ⊆ agent_view∧
           inconsistent(K_s,Dom(x_self),Dom
               backtrack(nogoods);
          end if;
        elseif x_selfstatus ≠ x_selfstatusold then
(c.6)     send (ok?(self,(x_selfstatus,d_selfstatus)))
          to connected constraint evaluating agents;
        end if;
(c.7) else nogoods←{K_s|K_s ⊆ agent_view∧
       inconsistent(K_s,Dom(x_selfstatus))};
              backtrack(nogoods);
      end if; end check_agent_view;
(d) procedure backtrack (nogoods);
      if ∅ ∈ nogoods then
        broadcast to other agents (¬∃ solution);
        terminate algorithm;
      end if;
      ∀ K_s ∈ nogoods do
        select a_k∈ agents(K_s):lowest priority (a_k);
        send (nogood(self,K_s)) to a_k;
        remove {(k,(x_k,d)),(k,(x_kstatus,d_status))}
                    from agent_view;
      end do; check_agent_view; end backtrack;
```

3.2 Example: Solving a DDCSP

In the following we give a simple example consisting of three agents a_1, a_2, and a_3 (see *Figure 2*). We define a set of variables $\{x_1,x_2,x_3\}$ belonging to the agents a_1, a_2, and a_3 where $Dom(x_1)=\{1,2\}$, $Dom(x_2)=\{3\}$, and $Dom(x_3)=\{2\}$.

We define sets of initially active variables: $V_{1_{start}}=\{x_1\}$, $V_{2_{start}}=\emptyset$, $V_{3_{start}}=\emptyset$. In order to store the variable state of agent variables we define $\{x_{1status}, x_{2status}, x_{3status}\}$, where $Dom\ (x_{1status}) = Dom\ (x_{2status}) = Dom\ (x_{3status}) = \{active, inactive\}$. Finally, we introduce the following activity constraints: C_{A31}: $(x_1=1)\overset{RV}{\Rightarrow}x_3$, C_{A21}: $(x_1)\overset{ARV}{\Rightarrow}x_2$, and C_{A22}: $(x_3)\overset{ARN}{\Rightarrow}x_2$, where C_{Aik} denotes activity constraint k of a_i. *Figure 3* shows four snapshots of the solving process.

Agent a_1 locally instantiates its variables without regarding the instantiations of remote agents: $x_1=1$. Now a_1 sends its variable instantiation to the constraint evaluating agents a_2 and a_3 , i.e. *ok?(1,(x_1,1))* . Agents a_2 and a_3

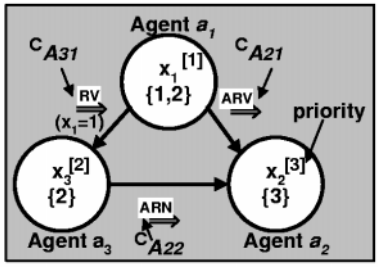

Fig. 2. Example: Distributed Dynamic CSP

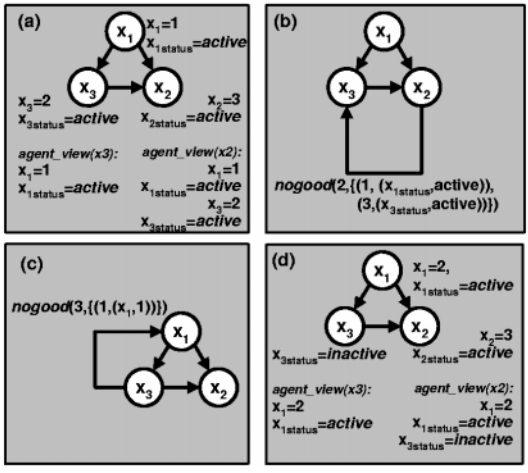

Fig. 3. Views on the solving process

store these values in their local *agent_view*. Both agent variables are activated, i.e. the state of x_2 and x_3 is changed to *active*. Now agent a_3 sends its variable instantiation to agent a_2, *i.e. ok?(3,*(x_3,2)). The *view (a)* of *Figure 3* represents the situation after agent a_2 has stored the value change of x_3 in its *agent_view*. Agent a_2 detects an inconsistency while checking its *Activity State Consistency*, since the activity constraints C_{A21} and C_{A22} are incompatible, i.e. x_2 can neither be activated nor deactivated. Agent a_2 determines those variables of the *agent_view* responsible for the inconsistent *Activity State* and communicates *nogood(2,*$\{(1,(x_{1status},active)), (3,x_{3status},active))\}$) to a_3, which is the lowest priority agent in the calculated *nogood (view (b) of Figure 3)*. This *nogood* is stored in the *nogoods_list* of a_3. Agent a_3 detects an inconsistent *Activity State*, since no value of $x_{selfstatus}$ is consistent with the *agent_view* and C_{A31}. The *nogood* $\{(1,(x_1,1))\}$ is sent to a_1 *(view (c) of Figure 3)*. Note that $(x_1,1) \Rightarrow (x_{1status},active)$.

Agent a_1 is in a consistent *Activity State*, i.e. no activity constraints are violated. For achieving *Agent Consistency* the value of x_1 is changed, i.e. x_1=2.

The new instantiation is propagated to a_2 and a_3. Subsequently a_2 and a_3 enforce *Activity State Consistency*. x_3 is deactivated *and ok?(3,($x_{3status}$,inactive))* is communicated to a_2. Agent a_2 removes *(3,(x_3,2))* from its *agent_view (view (d) of Figure 3)*.

3.3 Analysis

In order to show the soundness of *Algorithm 1* we must show that each generated solution (algorithm is in a stable state, i.e. all agents wait for incoming messages and no message is sent) satisfies the criteria stated in *Section 2*. *First*, the assignments satisfy each constraint, since each agent checks all local constraints after each value change in the local *agent_view*. *Second*, all variables $x_i \in V_{start}$ are active and instantiated in a solution, since $x_{istatus}$=*active* is a local constraint of agent a_i and all active variables are instantiated. Furthermore, x_i can not be deactivated, since $x_{istatus}$=*active* holds. *Third*, the minimality of the solution is guaranteed, since no variable has a solution relevant value, unless it is *active*. Each variable has well founded support either through $\overset{(A)RV}{\Rightarrow}$ constraints or through its membership in V_{start}. After each change in the local *agent_view*, the activity state of the variable is checked and updated. If no $\overset{(A)RV}{\Rightarrow}$ constraint is activated, the variables state is set to *inactive*, i.e. there is no further well founded support for the variable.

In order to show the completeness of the algorithm we must show that if a solution exists the algorithm reaches a stable state, otherwise the algorithm terminates with a failure indication (empty *nogood*). Let us first assume that the algorithm terminates. If this algorithm terminates by reaching a stable state then we have already shown that this is a correct solution. If this algorithm terminates by deducing the empty *nogood* then by applying resolution we detected that no consistent value assignment to the variables exists, i.e. no solution to the DDCSP exists. Finally we have to show that the algorithm terminates. Sources for infinite processing loops are cycles in message passing and subsequent searching of the same search space. Infinite processing loops are avoided because we require a total order of the agents. The *ok? (nogood)* messages are passed only from agents to connected agents with lower (higher) priority. *Nogoods* avoid subsequent searching of the same search space.

Dynamic constraint satisfaction is an NP-complete problem [13]. The worst-case time complexity of the presented algorithm is exponential in the number of variables. The worst-case space complexity depends on the strategy we employ to handle *nogoods*. The options range from unrestricted learning (e.g. storing all *nogoods*) to the case where *nogood* recording is limited as much as possible. For the agents in the presented algorithm it is sufficient to store only one *nogood* for each $d \in Dom(x_i)$. These *nogoods* are needed to avoid subsequent assignment of the same value for the same search space. In addition all *nogoods* can be deleted which contain a variable-value pair not appearing in the *agent_view*. Consequently the space-complexity for *nogood* recording is $O(n$ —D—), where n is the number of agents and —D— is the maximum cardinality of the domains. In addition the generation of *nogoods* is another source of high computational costs. Note, that in the presented algorithm it is sufficient to generate one *nogood*.

This *nogood* needs not be minimal, i.e. for the *nogood* generation in the procedure *backtrack* even the complete *agent_view* is an acceptable *nogood*. However, non-minimal *nogoods* lead to higher search efforts. The advantages and strategies for exploiting *nogoods* to limit the search activities are discussed in [1], [5].

4 Related Work

Different algorithms have been proposed for solving distributed CSPs. In [10] a distributed backtracking algorithm (DIBT) is presented which is based on the concept of graph based backjumping. The exploitation of *nogoods* is not supported. DIBT is especially powerful if combined with variable ordering techniques. Note that our presented algorithm can use any total ordering, which takes advantage of the actual problem structure and this algorithm also performs graph based backtracking. However, in practice variable ordering must take into account that sets of variables are assigned to one agent and that this assignment cannot be changed because of security or privacy concerns.

Asynchronous weak commitment search proposed by [14] employs a min-conflict heuristic, where a partial solution is extended by adding additional variables until a complete solution is found. In contrast to asynchronous backtracking all detected *nogoods* must be stored, in order to prevent infinite processing loops. In many configuration domains the problem size does not permit the storage of all generated *nogoods*, i.e. asynchronous backtracking is more applicable.

In [4] an agent architecture for solving distributed configuration-design problems employing an algorithm based on the concurrent engineering design paradigm is proposed. The whole problem is decomposed into sub-problems of manageable size which are solved by agents. The primary goal of this approach is efficient distributed design problem solving, whereas our concern is to provide effective support of distributed configuration problem solving, where knowledge is distributed between different agents having a restricted view on the whole configuration process.

5 Conclusions and Further Work

The integration of businesses by internet technologies boosts the demand for distributed problem solving. In particular in knowledge-based configuration we have to move from stand-alone configurators to distributed configuration. Dynamic constraint satisfaction is one of the most applied techniques in the configuration domain and therefore we have to extend this technique to distributed dynamic constraint satisfaction. In this paper we proposed a definition for distributed dynamic constraint satisfaction. Based on this definition we presented a complete and sound algorithm. This algorithm allows the exploitation of bounded learning algorithms. Configuration agents can exchange information about conflicting requirements (e.g. *nogoods*) thus reducing search efforts. The algorithm was implemented by using ILOG configuration and constraint solving libraries. The concepts presented in this paper are an essential part of an integrated environment for the development of cooperative configuration agents, where a concep-

tual model of configuration agents is automatically translated into an executable logic representation [8].

Further work will include additional applications of the presented algorithm to various configuration problems. These applications will help us to gain more insights in the nature of configuration problems thus providing the basis for further work on DDCSP strategies. E.g. applications in the telecommunication domain support the assumption that in configuration domains *nogoods* tend to be small in their arity. This suggests the application of constraint learning techniques to focus the search.

In addition we are investigating extensions of the basic dynamic CSP paradigm in order to include concepts such as disjunction or default negation as proposed by [13] and generative CSP representation [9].

References

1. R.J. Bayardo and D.P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings AAAI*, pages 298-304, Portland, Oregon, 1996.
2. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3,1:63-75, 1985.
3. R. Weigel D. Sabin. Product Configuration Frameworks - A Survey. In E. Freuder B. Faltings, editor, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13,4, pages 50-58. 1998.
4. T.P. Darr and W.P. Birmingham. An Attribute-Space Representation and Algorithm for Concurrent Engineering. *AIEDAM*, 10,1:21-35, 1996.
5. R. Dechter. Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence*, 40,3:273-312, 1990.
6. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353-366, 1989.
7. B. Faltings, E. Freuder, and G. Friedrich, editors. Workshop on Configuration. *AAAI Technical Report WS-99-05*, Orlando, Florida, 1999.
8. A. Felfernig, G. Friedrich, and D. Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. In *11th International Conference on Software Engineering and Knowledge Engineering*, pages 337-345, Kaiserslautern, Germany, 1999.
9. G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. In E. Freuder B. Faltings, editor, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13,4, pages 59-68. 1998.
10. Y. Hamadi, C. Bessiere, and J. Quinqueton. Backtracking in distributed Constraint Networks. In *Proceedings of ECAI* 1998, pages 219-223, Brighton, UK, 1998.
11. S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proceedings of AAAI 1990*, pages 25-32, Boston, MA, 1990.
12. F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of ECAI 1990*, Stockholm, Sweden, 1990.
13. T. Soininen, E. Gelle, and I. Niemelä. A Fixpoint Definition of Dynamic Constraint Satisfaction. In *5th International Conference on Principles and Practice of Constraint Programming - CP'99*, pages 419-433, Alexandria, USA, 1999.
14. M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem. *IEEE Transactions on Knowledge and Data Engineering*, 10,5:673-685, 1998.
15. M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS-98)*, Paris, pages 372-379, 1998.