Distributed generative CSP approach towards multi-site product configuration

A. Felfernig[1], G. Friedrich[1], D. Jannach[1], M.C. Silaghi[2], and M. Zanker[1]

[1] Universitaet Klagenfurt,
A-9020 Klagenfurt, Austria
{felfernig,friedrich,jannach,zanker}@ifit.uni-klu.ac.at
[2] Florida Institute of Technology (FIT),
Melbourne, FL 32901, USA
msilaghi@cs.fit.edu

Abstract. Today's configurators are centralized systems and do not allow manufacturers to cooperate on-line for offer-generation or sales-configuration. However, supply chain integration of configurable products requires the cooperation of the configuration systems from the different manufacturers that jointly offer solutions to customers. As a consequence, there is a business requirement for methods that enable the computation of such configurations by independent specialized agents. Some of the approaches to *centralized* configuration tasks are based on constraint satisfaction problem (CSP) solving. Most of them extend the traditional CSP approach in order to answer the specific expressivity and dynamism requirements for configuration and similar synthesis tasks.
The distributed generative CSP (DisGCSP) framework proposed here builds on a CSP formalism that encompasses the *generative* aspect of variable creation and of extensible domains for problem variables. It also builds on the distributed CSP (DisCSP) framework, allowing for approaches to configuration tasks where the knowledge is distributed over a set of agents. Notably, the notions of a *constraint* and a *nogood* are generalized to an additional level of abstraction, extending inferences to types of variables. The usage of the new framework is exemplified by describing modifications to asynchronous algorithms. Our experimental evaluation gives evidence that for typical configuration tasks, encodings within a DisGCSP framework can be solved more efficiently than encodings with DisCSP.

1 Introduction/Background

The paradigm of mass-customization allows customers to tailor (configure) a product or service according to their specific needs, i.e., the customer can select between several features and options that should be included in the configured product and can determine the physical component structure of the personalized product variant. Typically, there are several technical and marketing restrictions on the legal parameter constellations and on the physical layout. This led manufacturers to develop support for checking the feasibility of user requirements and for computing a consistent solution. Such functionality is provided by product configuration systems (configurators), which constitute a successful application area for different Artificial Intelligence techniques [27], e.g. description logics [17] or rule-based [2] and constraint-based solving algorithms. [8]

describes the industrial use of constraint techniques for the configuration of large and complex systems such as telecommunication switches and [16] details an example of a powerful tool based on Constraint Satisfaction available on the market.
However, companies find themselves in dynamically determined coalitions with other highly specialized solution providers that jointly offer customized solutions. This high integration aspect of today's digital markets requests that software products supporting the selling and configuration tasks are no longer conceived as standalone systems. A product configurator can be therefore seen as an agent with private knowledge that acts on behalf of its company and cooperates with other agents to solve configuration tasks. This paper abstracts the *centralized* definition of configuration tasks in [28] to a more general definition of a *generative* CSP that is also applicable to the wider range of synthesis problems. Furthermore, we propose a framework that allows us to address distributed configuration tasks by extending DisCSPs with the innovative aspects of local generative CSPs:

1. The constraints (and nogoods) are generalized to a form where they can depend on types rather than on identities of variables. This also enables an elegant treatment of the following aspects.
2. The number of variables of certain types that are active in the local Generative CSP (GCSP) of an agent, may vary depending on the state of the search process. In the DisCSP framework, the external variables existing in the system are predetermined, but in a DisGCSP the set of variables defining the problem is determined dynamically.
3. The domain of the variables may vary dynamically. Some variables model possible connections and they depend on the existence of components that could become connected. Namely, these domains extend when the possibility of connection to new components is created.

We describe the interesting impact of the previously mentioned changes on asynchronous algorithms. After giving a motivating example in Section 2, Section 3 defines a generative CSP. Section 4 formalizes a distributed generative CSP and in Section 5 extensions to current DisCSP frameworks are presented. Finally, Section 6 evaluates DisGCSP encoding vs. classic DisCSP problem representation for typical configuration problems.

2 Example of configuration problem

In the following we exemplify a problem from the domain of product configuration ([8]). A telecommunication switch consists of a set of frames, where each frame has optional connection points (denoted as ports) for modules to be plugged in. These modules can be configured to be either *analog* or *digital* (Figure 1). In addition, problem-specific constraints describe legal combinations of the number and configuration of the modules on the different frames. As different companies have to cooperate to provide a switching solution, the distribution aspect is inherent in this scenario. In order to keep the example simple, we assume a two-agent setting. The agents A_1 and A_2 are capable of configuring different functionalities of the telecommunication switch and therefore each of them owns a different set of constraints and they have a limited view on the

overall system, e.g., agent A_1 requires only a view on the configuration of the upper two frames.

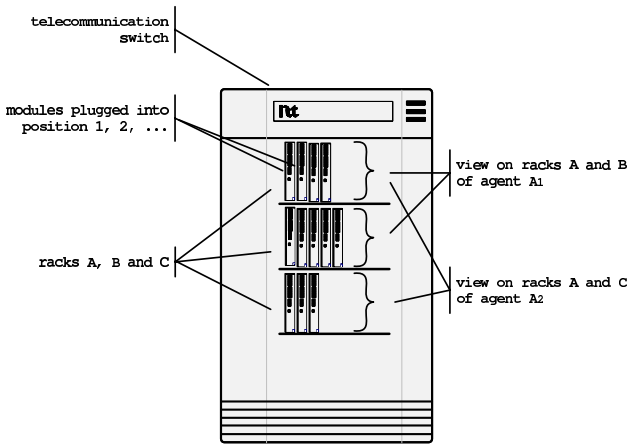

Fig. 1. Telecomunication switch

Short introduction on encoding technique When configuring large technical systems, the exact number of problem variables, i.e. employed modules and their connections, is not known from the beginning, it is therefore not efficient to formulate constraints directly on concrete variables.
Instead, comparable to programming languages, variable types exist that allow us to associate a newly created variable with a domain and we can specify relationships in terms of *generic constraints*. [28] defines a generic constraint γ as a constraint schema, where meta-variables M_i^t act as placeholders for concrete variables of a specific type t, denoted by the superscript t. The subscript i allows us to distinguish between different meta-variables in one constraint[3]. We therefore formalize this configuration task as a CSP, where for each frame two different types of variables exist. The first one encompasses the variables that represent the ports of a frame and the second one the modules to which the ports can be connected, i.e. t_{pa}, t_{pb}, t_{pc} denote the types of port variables of frame A, B and C and t_a, t_b, t_c represent the types of module variables. Every single port and module is represented by a variable[4]. Variables representing modules take either the value 0 (analog) or 1 (digital), while port variables are assigned the index value of the module they are connected to. In our example for denoting a single variable the naming convention $type_i$ is chosen, where the subscript i gives the index value of variables of the same type, e.g., $a_1, a_2, \ldots$ represent the module instances on frame A and

[3] The exact semantics of generic constraints is given in Definition 3 in Section 3.1

[4] Note, that the *frame* components themselves are not explicitly modeled, but only via their characterizing port variables.

$pa_1, pa_2, \ldots$ are the names of its port variables.
In order to be able to formulate restrictions on the amount of variables for each type a counter variable (x_{type}) exists that holds the number of instantiated variables of type $type$.
The configuration constraints are distributed between two agents, i.e., each agent A_i posesses a set of local constraints[5] Γ^{A_i}, i.e., $\Gamma^{A_1} = \{\gamma_1, \gamma_2, \gamma_4, \gamma_6\}$ and $\Gamma^{A_2} = \{\gamma_3, \gamma_5, \gamma_7, \gamma_8\}$, which are defined as follows:
Variables representing modules can take either the value 0 ('analog') or 1 ('digital').
$\gamma_1 : M^{t_a} \leq 1$. $\gamma_2 : M^{t_b} \leq 1$. $\gamma_3 : M^{t_c} \leq 1$.
For agent A_1 the modules on frame A and those on frame B must be configured differently, i.e. all modules on frame A are set as 'analog' and all modules on frame B as 'digital' or vice versa.
$\gamma_4 : M_1^{t_a} \neq M_2^{t_b}$.
For agent A_2 the modules on frame A and on frame C must have the same configuration, i.e. they must be all set as either 'analog' or 'digital'.
$\gamma_5 : M_1^{t_a} = M_2^{t_c}$.
Agent A_1 ensures, that the amount of modules on frame A is equal to the amount of modules on frame B.
$\gamma_6 : x_{t_a} = x_{t_b}$., where x_{t_a} resp. x_{t_b} is a counter variable, that holds the amount of instantiated module variables on frame A resp. frame B.
Similarly for agent A_2, the amount of modules on frame C must be at least as high as the amount of modules on frame A minus 1.
$\gamma_7 : x_{t_a} - 1 \leq x_{t_c}$.
Agent A_2 ensures that all modules on frame C are configured as 'digital'.
$\gamma_8 : M^{t_c} = 1$.

Example of the usage of the framework Now, we will exemplify the solving process in our framework for distributed generative constraint satisfaction explained later on this paper. Solving is performed by our asynchronous forward checking algorithm with local constraints (compare trace in Figure 3), that is detailed in Section 4. The subscripted index value of the agents A_i also denotes their priority, where 1 is highest. The example in Figure 2.a depicts an initial situation, where a customer-specific requirement imposes a restriction on the configuration result, e.g. the telecommunication switch must contain at least two modules that must be connected via ports to frame A. Agent A_1 fulfills this initial customer requirement by generation of problem variables and communicates them via an **announce** message to the other Agent A_2. The parameter of an **announce** is a list of new variables denoted by a pair: (type,index). Agent A_2 determines his interest in the newly announced variables and communicates an **addlink** message back. As can be seen from Figure 2.b, agent A_1 creates two modules and their ports on frame B in order to fulfill constraint γ_6 and sets modules on frame A to 0 and on frame B to 1 (constraint γ_4). Agent A_2 creates a module instance on frame C and configures it

[5] For reasons of presentation we omit those constraints that ensure that all modules are connected and that once a port variable is assigned a value, the corresponding connected component variable must exist.

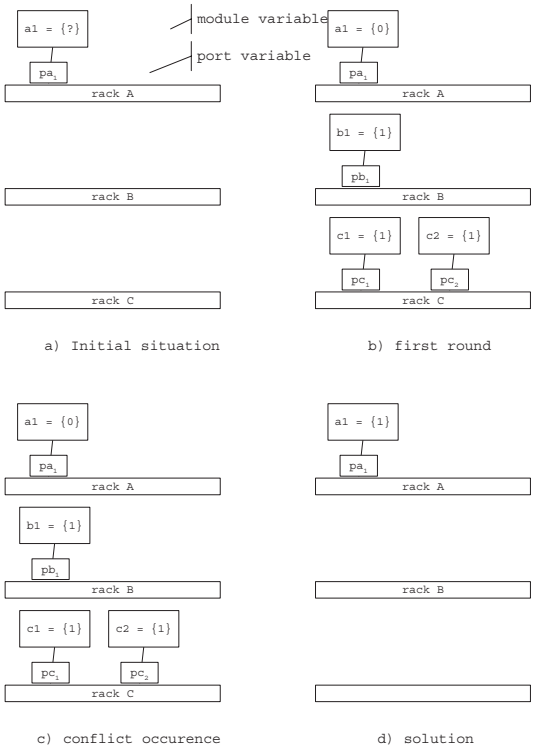

Fig. 2. Example problem

1: $A_1 \xrightarrow{\textbf{announce}(\langle t_a, 1\rangle, \langle t_a, 2\rangle, \langle t_{pa}, 1\rangle, \langle t_{pa}, 2\rangle)} A_2$
2: $A_2 \xrightarrow{\textbf{addlink}\langle\{(t_a, 1), (t_a, 2), (t_{pa}, 1), (t_{pa}, 2)\}\rangle} A_1$
3: $A_1 \xrightarrow{\textbf{announce}(\langle t_b, 1\rangle, \langle t_b, 2\rangle, \langle t_{pb}, 1\rangle, \langle t_{pb}, 2\rangle)} A_2$
4: $A_2 \xrightarrow{\textbf{announce}(\langle t_c, 1\rangle, \langle t_{pc}, 1\rangle)} A_1$
5: $A_1 \xrightarrow{\textbf{ok?}\langle M^a, \{1, 2\}, 0\rangle} A_2$
6: $A_1 \xrightarrow{\textbf{ok?}\langle M^{pa}, \{1\}, 1\rangle} A_2$
7: $A_1 \xrightarrow{\textbf{ok?}\langle M^{pa}, \{2\}, 2\rangle} A_2$
8: $A_2 \xrightarrow{\textbf{nogood}\neg(\langle M^a, \{1, 2\}, 0\rangle)} A_1$
9: $A_1 \xrightarrow{\textbf{ok?}\langle M^a, \{1, 2\}, 0\rangle} A_3$

Fig. 3. Trace of the solving process with AFC$_c$-GCSP

following constraint γ_8 as 'digital' ($c_1 = 1$).
In order to reflect the interchangeability of value assignments, generic assignments using a meta-variable, a set of index values and their assigned value are exchanged in an **ok?** message. When agent A_1 communicates the assignments to module variables a_1 and a_2 to agent A_2 an inconsistency is detected by agent A_2 (violation of constraint γ_5). As agent A_2 is not capable of locally resolving this conflict by changing his local assignments, he calculates a nogood and communicates it back to agent A_1 (Figure 2.c). Note, that we use here generic nogoods that exploit the interchangeability of the conflicting variables, e.g., all variables of type t_a must not take the value 0. Agent A_1 resolves the conflict by changing its variable assignments and finally a solution is found (Figure 2.d). In this simple example, the generic nogood involved only one type, but in general it can involve several variable types.
Consequently, a solution to a generative constraint satisfaction problem requires not only finding valid assignments to variables, but also determining the exact size of the problem itself. In the sequel of the paper we define a model for a Generative constraint satisfaction problem of a local configurator and detail then the extensions to an asynchronous algorithms in order to be applicable for DisGCSPs.

3 Generative Constraint Satisfaction

In many applications, solving is a *generative* process, where the number of involved components (i.e., variables) is not known from the beginning. To represent these problems we employ an extended formalism that complies to the specifics of configuration and other synthesis tasks. For efficiency reasons, problem variables representing components of the final system are *generated* dynamically as part of the solution process because their exact number cannot be determined beforehand. The framework is called *generative* CSP (GCSP) [10, 28]. This kind of dynamicity extends the approach of dynamic CSP (DCSP) formalized by Mittal and Falkenhainer [19], where all possibly involved variables have to be known from the beginning, since the activation constraints reason on the variable's activity state. [20] propose a conditional CSP to model a configuration task, where structural dependencies in the configuration model are exploited to trigger the activation of subproblems. Another class of DCSP was first introduced by [6] where constraints can be added or removed independently of the initial problem statement. A GCSP is extended in order to find a consistent solution while the DCSP in [6] has already a solution and is extended due to influence from the outside world (e.g., additional constraints) that necessitates finding a new solution. Here we give a definition of a GCSP that abstracts from the specifically for configuration tasks formulated approach in [28] and applies to the wider range of synthesis problems.
Exploiting incremental relaxation of the unary constraints in CSPs (as we introduce in GCSP) is a promising technique for approaching any CSP. To formally refer to states reached along this relaxation, a first attempt could consider the framework given by the following definition [7, 26]:

Definition 1 (Open Constraint Satisfaction Problems). *An Open CSP (OCSP), O, is defined as a set of CSPs with the same variables and constraints*[6]*, $\langle O(1), O(2), ...\rangle$. A partial order, $\prec$, is defined on O. $O(i) \prec O(j)$ when all the values in any domain of $O(i)$ are also present in the corresponding domain of $O(j)$. $O(i) \prec O(j)$ implies $i < j$. Excepting $O(1)$, all other CSPs in O have a predecessor: $\forall j > 1, \exists i, O(i) < O(j)$.*

The relation between an OCSP and a Generative CSP will be discussed in more depth in Section 3.2.

3.1 Generative Constraint Satisfaction

Definition 2 (Generative constraint satisfaction problem (GCSP)). *A generative constraint satisfaction problem is a tuple GCSP(X_0, Γ, T, Δ_0), where:*

- *X_0 is the set of initially given variables.*
- *Γ is the set of generic constraints.*
- *$T = \{t_1, \ldots, t_n\}$ is the set of variable types t_i, where $dom(t_i)$ associates the same domain to each variable of type t_i, where the domain is a set of atomic values.*
- *For every type $t_i \in T$ there exists a counter variable $x_{t_i} \in X_0$ that holds the number of variable instantiations for type t_i. Thus, explicit constraints involving the total number of variables of specific types and reasoning on the size of the CSP becomes possible. The set of counter variables is C.*
- *Δ_0 is a relation on $X_0 \setminus C \times (T, N)$, where N is the set of positive integer numbers. Each tuple $(x, (t, i))$ associates a variable $x \in X \setminus C$ with a unique type $t \in T$ and an index i, that indicates x is the i^{th} variable of type t. The function $type(x)$ accesses Δ_0 and returns the type $t \in T$ for x and the function $index(x)$ returns the index of x.*

Definition 3 (Generic constraint). *A meta-variable M_i is associated a variable type $type(M_i) \in T$ and must be interpreted as a placeholder for all concrete variables x_j, where $type(x_j) = type(M_i)$. A generic constraint $\gamma \in \Gamma$ formulates a restriction on the meta-variables $M_a, \ldots, M_k$, and eventually on counter variables.*

By generating additional variables and relaxing unary constraints[7], a previously unsolvable CSP can become solvable, which is explained by the existence of counter variables that hold the number of variables. To concisely specify that a meta-variable M_i has type t, we denote it by M_i^t.

When modeling a configuration problem, variables representing named connection points between components, i.e., *ports*, will have references to other components as their domain. Consequently, we need variables whose domain varies depending on the size of a set of specific variables [28].

Example Given t_a as the type of variables representing *modules* and t_{pa} as the type of *port* variables that are allowed to connect to *modules*, then the domain of the *pa*

[6] [31] introduces the notion of constraints that can be known without knowing domains, as a source of privacy in ABT.

[7] Which can be dually seen as addition of values.

variables $dom(t_{pa})$ must contain references to *modules*. This is specified by defining $dom(t_{pa}) = \{0, \ldots, \infty\}$, by a meta-constraint $M_1^{t_{pa}} < x_{t_a} + 1$, and by formulating an additional generic constraint that restricts all variables of type t_{pa} using the counter variable for the total number of variables having type t_a, i.e., $M_1^{t_{pa}} < x_a + 1$. With the help of the $index(x)$ function concrete variables can also then be referenced.
We can formalize the local GCSP of an agent A_1 in the initial situation as $X_0^{A_1} = \{x_a, x_{pa}, x_b, x_{pb}, a_1, a_2, pa_1, pa_2\}$,
$\Gamma^{A_1} = \{\gamma_1, \gamma_2, \gamma_4, \gamma_6\}$, $T^{A_1} = \{t_a, t_{pa}, t_b, t_{pb}\}$ and $\Delta^{A_1} = \{(a_1, (t_a, 1)), (a_2, (t_a, 2)), (pa_1, (t_{pa}, 1)), (pa_2, (t_{pa}, 2))\}$. The $index(a_1)$ function returns 1, which indicates that a_1 is the first module instance on frame A. The domains of variables are consequently defined as $dom(t_a) = dom(t_b) = \{0, 1\}$ and $dom(t_{pa}) = dom(t_{pb}) = \{0, \ldots, \infty\}$, which can be additionally limited by domain constraints (e.g., $\gamma_1 : M^{t_a} \leq 1$.).
Note, that generic constraints can also formulate restrictions on specific initial variables from X_0 by employing the $index()$ function.
Consider the GCSP(X, Γ, T, Δ) and let $\gamma \in \Gamma$ restrict the meta-variables $M_a, \ldots, M_k$, where $type(M_i) \in T$ is the defined variable type of the meta variable M_i. The consistency of generic constraints is defined as follows:

Definition 4 (Consistency of generic constraints). *Given an assignment tuple θ for the variables X, then γ is said to be satisfied under θ, iff* $\forall x_a, \ldots, x_k \in X \setminus C$: $type(x_a) = type(M_a) \wedge \ldots \wedge type(x_k) = type(M_k) \rightarrow \gamma[M_a|_{x_a}, \ldots, M_k|_{x_k}]$ *is satisfied unter θ, where $M_i|_{x_i}$ indicates that the meta-variable M_i is substituted by the concrete variable x_i.*

Thus a *generic* constraint must be seen as a constraint scheme that is expanded into a set of constraints after a preprocessing step, where meta-variables are replaced by all possible combinations of concrete variables having the same type. For example, given a fragment of a GCSP of agent A_1 with $X^{A_1} = \{a_1, a_2, b_1, b_2\}$, $T^{A_1} = \{t_a, t_b\}$ and $\Delta^{A_1} = \{(a_1, (t_a, 1)), (a_2, (t_a, 2)), (b_1, (t_b, 1)), (b_2, (t_b, 2))\}$, the satisfiability of the generic constraint γ_4 is checked by testing the following conditions: $a_1 \neq b_1$. $a_1 \neq b_2$.$a_2 \neq b_1$.$a_2 \neq b_2$.
Consequently, reasoning over a GCSP consists in generating new variables and values and in expanding generic constraints into concrete constraints on these variables. At each moment the currently generated elements can be used to define a CSP.

Definition 5 (State of search in GCSP). *The state of search in a GCSP(X_0, Γ, T, Δ_0) at moment t is a CSP, CSP($X(t)$,$P(t)$,D). $X(t) = X'(t) \cup X_0$ where $X'(t)$ is a set of variables generated with types T, based on the GCSP. $P(t)$ is a set of constraints inferred from the instantiation of Γ with $X(t)$, namely it also includes nogoods and the labels (constraint on possible values) for variables. D is the domain of the variables $X(t)$ as given by their types, and static.*

Note that generic constraints break a special type of symmetries, namely where several variables are interchangeable. This is a hot topic and other recent results confirm the important efficiency gains offered by exploitation of symmetries [9]. Most approaches try to break symmetries as a means of reducing complexity, and the generic constraints and nogoods can be seen as a powerful symmetry breaker.

Definition 6 (Solution for a generative CSP). *Given a generative constraint satisfaction problem GCSP(X_0, Γ, T, Δ_0), P, then its solution encompasses the finding of a set of variables X, type and index assignments Δ, defining a state of search in P, and an assignment tuple θ for the variables in X, s.t.*

1. *for every variable $x \in X$ an assignment $x = v$ is contained in θ, s.t. $v \in dom(type(x))$ and*
2. *every constraint $\gamma \in \Gamma$ is satisfied under θ and*
3. *$X_0 \subseteq X \wedge \Delta_0 \subseteq \Delta$.*

When solving a GCSP in the context of practical applications, we do not impose a minimality criterion on the number of variables in our solution, because different optimization criteria exist, such as total cost or flexibility of the solution, thus non-minimal solutions can be preferred over minimal ones.
Note, that names for generated variables are unique and can be randomly chosen by the GCSP solver implementation and therefore constraints must not formulate restrictions on the variable names of generated variables. Consequently, substitution of any generated variable (i.e., $x \in X \setminus X_0$) by a newly generated variable with equal type, index and value assignment has no effect on the consistency of generic constraints.

3.2 Comparing GCSPs and OCSPs

A GCSP can be seen as a specialized OCSP, namely where a strict relation is enforced on the values added in one step to different variables. This relation is a knowledge belonging to the configuration domain. The inexisting variables correspond then to variables currently having empty domains in OCSPs.

Theorem 1. *For any GCSP, there exists an OCSP that is satisfiable for the same state.*

Proof By projecting the constraint obtained by combining all the constraints of the states of the GCSP onto its counter variables, one obtains an OCSP since each state is obtained by adding additional values to the counter variables.

Theorem 2. *For any OCSP, there exists an GCSP that is satisfiable for the same state.*

Sketch of the proof. For any OCSP, one can design a GCSP where each addition of a value is associated to the addition of a variable. The constraints of the OCSP are translated into meta-constraints of corresponding types. Such constraints can be designed for obtaining any wished combination of needed components.

This relation does not apply to solutions, since in the proof of Theorem 1 the OCSP has no knowledge on the generated variables. We can potentially use OCSPs for configuration with a dynamic set of partners found on the web, but that is a longer term research that we have not yet fully solved.

3.3 Comparing GCSP and generative constraint based configuration

Our GCSP definition extends the definition from [28] in the sense that a finite set of variable types T is given and during problem solving variables having any of these types can be generated, whereas in [28] only variables of a single type, i.e., component variables, can be created. Current CSP implementations of configuration systems (e.g., [16] [8]) use a type system for problem variables, where new variable instances, having one of the predefined types, are dynamically created. This is only indirectly reflected in the definition of [28] by the domain definition of component variables, which we explicity represent here as a set of types. Furthermore, the definition of *generic constraints* does not enforce the use of a specific constraint language[8] for the formulation of restrictions.

3.4 Comparing GCSP and DCSP

The set of variables X could be theoretically infinite, leading to an infinite search space. For practical reasons, solver implementations for a GCSP put a limit on the total number of problem variables to ensure decidability and finiteness of the search space. This way a GCSP is reduced to a dynamic CSP and in further consequence to a CSP. A DCSP models each search state as a static CSP, where complex activation constraints are required to ensure the alternate activation of variables depending on the search state. These constraints need to be formulated for every possible state of the GCSP, which leads to combinatorial explosion of concrete constraints. Furthermore, the formulation of large configuration problems as a DCSP is merely impractical from the perspective of knowledge representation, which is crucial for knowledge-based applications such as configuration systems.

4 Distributed Constraint Satisfaction

Algorithms for configuration applications need to ensure a good/optimal solution, that's why we focus on complete algorithms in our framework. The first asynchronous complete search algorithm is Asynchronous Backtracking (ABT) [30]. An enhanced version for several variables per agent is described in [32]. [3] show how ABT can be adapted to networks where not all agents can directly communicate to one another. [11] makes the observation that versions of ABT with polynomial space complexity can be designed. Extensions of ABT with asynchronous maintenance of consistencies, and asynchronous dynamic reordering are described in [29, 1, 22, 25]. [21] achieves an increased level of abstraction in DisCSPs by letting nogoods (i.e., certain constraints) consist of aggregates (i.e., sets of variable assignments), instead of simple assignments.
Asynchronous Backtracking (ABT) takes its name from the fact that the *operations* performed by each agent resemble backtracking:

– A value that is compatible with all predecessor variables is chosen at each step.

[8] Examples are the LCON language used in the COCOS project [28], or the configuration language of the ILOG Configurator [16].

As reported in [18], one tries later to implement asynchronous version for forward checking by letting each agent to perform the *operations* of forward checking:

– At each step a value is chosen and domains of successors are pruned.

It is noticed in [18] that the obtained version, Distributed Forward Checking (DFC), did not bring the expected improvements. Besides the details described in [3], a plausible explanation of the comparison DFC vs. ABT is that despite differences in operations performed by each agent, the two algorithms perform very similarly from a *phenomenological* point of view:

The pruning performed with DFC is actually also performed with ABT (after one message trip) by the agent owner of the successor variables. This is done either implicitely as in [30], or even explicitely with a stack [15].
We can conclude that ABT is, *phenomenologically*, an asynchronous version of Forward Checking.

Based on the aforementioned properties of ABT, we develop a framework for solving distributed configuration tasks, where constraints are kept private for security reasons of the involved agents. Consequently, we propose an algorithm for problem solving termed Asynchronous Forward Checking with local constraints (AFC_c). This algorithm is an adaptation of ABT to the usage of private constraints (see Algorithm 1). The name, AFC, is chosen as a repair and an explanation of the behavior of ABT. We summarize in the following the properties of the AFC_c algorithm that guarantee its correctness and completeness [30].

4.1 Properties of AFC_c

We summarize the characteristics of most asynchronous search algorithms, reformulated to allow agents to know only the constraints that they enforce. They are considered as follows:

1. $A = \{A_1, \ldots A_n\}$ is a set of n totally ordered agents, where A_i has priority over A_j if $i < j$.
2. Each agent A_i owns a set of private constraints and has a *view* (will be defined in the following) on all variables that he is interested in. The constraints known by A_i are referred to as its local constraints, denoted Γ^{A_i} and A_i is *interested in* those variables that are contained in its local constraints. A *link* exists between two agents if they share interest in the same variable, that is directed from the agent with higher priority to the agent with lower priority. A link from agent A_1 to agent A_2 is referred to as an *outgoing link* of A_1 and an *incoming link* of A_2.
3. An assignment is a pair (x_j, v_j), where x_j is a variable, and v_j a value for x_j.
4. The *view* of an agent A_i is a set of the most recent assignments received for those variables agent A_i is interested in.
5. The agents communicate using the following types of messages, where channels without message loss are assumed:
   – **ok?** message. An agent in AFC_c proposes an assignment only to variables in which no higher priority agent is interested in. He then communicates via **ok?** messages each proposed assignment to lower priority agents.

- **nogood** message. In case an agent cannot find assignments that do not violate its own constraints and its stored nogoods, it generates an explanation under the form of an explicit nogood $\neg N$. A nogood can be interpreted as a constraint that forbids a combination of value assignments to a set of variables. It is announced via a **nogood** message to the lowest priority agent that has proposed an assignment in N.
- **addlink** message. The receiver agent is informed that the sender is interested in its variable. A *link* is established from the higher priority agent to the agent with lower priority.

Improving the performance of AFC_c with extensions as referenced in the introduction of this section is therefore straightforward. In the next section we apply this DisCSP framework with private constraints to a scenario where each agent locally solves a generative constraint satisfaction task.

5 Framework for DisGCSP

The DisCSP framework from the previous section will now be applied to a scenario of distributed product configuration. A distributed configuration problem is a multi-agent scenario, where each agent wants to satisfy a local GCSP and agents keep their constraints private for security and privacy reasons, but share all variables which they are interested in. First the *interest* of an agent in variables needs to be redefined, as generic constraints employ meta-variables:

An agent A_j owning a local $GCSP^{A_j}(X^{A_j},\Gamma^{A_j},T^{A_j},\Delta^{A_j})$ is said to be *interested in a typed-variable* $x \in X^{A_h}$ of an agent A_h, if there exists a generic constraint $\gamma \in \Gamma^{A_j}$ formulating a restriction on the meta-variables $M_a,\ldots,M_k$, where $type(M_i) \in T^{A_j}$ is the defined variable type of the meta variable M_i, and $\exists M_i \in M_a,\ldots,M_k : type(x) = type(M_i)$. An agent A_j owning a local $GCSP^{A_j}(X^{A_j},\Gamma^{A_j},T^{A_j},\Delta^{A_j})$ is said to be *interested in a counter-variable* $x_t \in X^{A_h}$ of an agent A_h, if there exists a generic constraint $\gamma \in \Gamma^{A_j}$ formulating a restriction on the counter variable x_t or on a variable of type t.

Definition 7 (Interest in variables). *An agent A_j owning a local $GCSP^{A_j}(X^{A_j},\Gamma^{A_j},T^{A_j},\Delta^{A_j})$ is said to be interested in a variable $x \in X^{A_h}$ of an agent A_h, if x is a counter-variable and A_j is interested in the counter-variable x, or otherwise if A_j is interested in the typed-variable x.*

Definition 8 (Distributed generative CSP). *A distributed generative constraint satisfaction problem has the following characteristics:*

- *$A = \{A_1,\ldots,A_n\}$ is a set of n agents, whereby each agent A_i enforces an original $GCSP_o^{A_i}(X_o^{A_i},\Gamma_o^{A_i},T_o^{A_i},\Delta_o^{A_i})$, and a cumulated[9] $GCSP_c^{A_i}(X_c^{A_i},\Gamma_c^{A_i},T_c^{A_i},\Delta_c^{A_i})$. The original and the cumulated GCSP of an agent form together its local GCSP $GCSP^{A_i}(X_o^{A_i}\cup X_c^{A_i},\Gamma_o^{A_i}\cup\Gamma_c^{A_i},T_o^{A_i}\cup T_c^{A_i},\Delta_o^{A_i}\cup\Delta_c^{A_i})$.*

[9] Received via **nogood** messages.

```
when received (ok?,⟨x_j, d_j⟩) do
    add ⟨x_j, d_j⟩ to agent view;
    check_agent_view;
when received (nogood,A_j,¬N) do
    when ⟨x_k, d_k⟩, where x_k is not connected, is contained in ¬N
        send addlink to owner(x_k);
        add ⟨x_k, d_k⟩ to agent view;
    put ¬N in nogood-list;
    when have variable
        old_value ← current_value;
    check_agent_view;
    when (have variable) and (old_value = current_value)
        send (ok?,⟨x_i, current_value⟩) to A_j;
procedure check_agent_view do
    if have variable then
        when agent view and current_value are not consistent
            if no value in D_i is consistent with agent view then
                backtrack;
            else
                select d ∈ D_i where agent view and d are consistent;
                current_value ← d;
                send (ok?,⟨x_i, d⟩) to lower priority agents in outgoing links;
    else
        when agent view is not consistent
            backtrack;
procedure backtrack do
    nogoods ← {V | V = inconsistent subset of agent view};
    when an empty set is an element of nogoods
        broadcast to other agents that there is no solution;
        terminate this algorithm;
    for every V ∈ nogoods do
        select ⟨x_j, d_j⟩ where x_j has the lowest priority in V;
        send (nogood,A_i,V) to owner(x_j);
        remove ⟨x_j, d_j⟩ from agent view;
    check_agent_view;
```

Algorithm 1: Procedures of A_i for receiving messages in AFC$_c$. The algorithm does not include a stable-state detection. In order to solve this task, one needs stable state detection algorithms like distributed snapshots [5] or any of the techniques in [26].

– *All variables in* $\bigcup_{i=1}^{n} X^{A_i}$ *and all type denominators in* $\bigcup_{i=1}^{n} T^{A_i}$ *share a common namespace, ensuring that a symbol denotes the same variable, resp. the same type, with every agent.*

- *For every pair of agents $A_i, A_j \in A$ and for every variable $x \in X^{A_j}$, where agent A_i is interested in x, must hold $x \in X^{A_i}$.*
- *For every pair of agents $A_i, A_j \in A$ and for every shared variable $x \in X^{A_i} \cap X^{A_j}$ the same type and index must be associated to x in the local GCSPs of the agents, i.e., $type^{A_i}(x) = type^{A_j}(x) \wedge index^{A_i}(x) = index^{A_j}(x)$.*
- *A function $owners(t)$ is defined for a type t, that returns a set of agents $A_{owners} \subseteq A$, where $\forall A_j \in A_{owners}$ holds $t \in T^{A_j}$.*

Consequently, for every pair of agents $A_i, A_j \in A$ and for every shared variable $x \in X^{A_i} \cap X^{A_j}$ a *link* must exist that indicates that they share variable x. The *link* must be directed from the agent with higher priority to the agent with lower priority.

Definition 9. *Given a distributed generative constraint satisfaction problem among a set of n agents then its solution encompasses the finding of a set of variables $X = \bigcup_{i=1}^{n} X^{A_i}$, type and index assignments $\Delta = \bigcup_{i=1}^{n} \Delta^{A_i}$ and an assignment tuple $\theta = \bigcup_{i=1}^{n} \theta^{A_i}$ for every variable in X, s.t. for all agents $A_i : X^{A_i}$, Δ^{A_i} and θ^{A_i} are a solution for the local $GCSP^{A_i}$ of agent A_i.*

Remark. A solution to a distributed generative CSP is also a solution to a centralized GCSP($\bigcup_{i=1}^{n} X^{A_i}, \bigcup_{i=1}^{n} \Gamma^{A_i}, \bigcup_{i=1}^{n} T^{A_i}, \bigcup_{i=1}^{n} \Delta^{A_i}$).

Definition 10 (Generic assignment). *A generic assignment is a unary generic constraint. It takes the form: $\langle M, i, v \rangle$, where M is a meta-variable, i is a set of index values for which the constraint applies, and v is a value.*

Definition 11 (Generic nogood). *A generic nogood takes the form $\neg N$, where N is a set of generic assignments for distinct meta-variables.*

5.1 Inferring generic nogoods

The techniques that can be applied for inferring generic nogoods are multiple and in general depend on the problem domain. General techniques for detecting and breaking symmetries on variables can be used straightforwardly. A similar technique applies to our example problem and was also exploited in the evaluation described next. We base the computation of nogoods (minimal conflicts) on Junker's QUICKXPLAIN algorithm [13], a non-intrusive conflict-detector that gets its efficiency by recursively partitioning the problem into subproblems of half the size and skipping those that do not contain an element of the propagation-specific conflict[10]. In the currently evaluated version, we only compute one, but minimal, conflict in each backtracking step, future work will include the computation of several conflicts at a time. A computed conflict both contains information on the amount of variables involved in the conflict as well as inconsistent variable assignments, whereby we can detect when the inconsistency arises solely from variable cardinalities which further improves the distributed search performance.

[10] Note, that we are only interested in propagation-specific conflicts that are induced by the values in the agent view.

5.2 Constraint relaxation for DisGCSPs

Value assignments to variables are communicated to agents via **ok?** messages that transport *generic assignments* in our DisGCSP framework, which represent domain restrictions on variables by unary constraints. Each of these unary constraints in our DisGCSP has attached an unique identifier called constraint reference (CR). Any inference has to attach the CRs associated to arguments into the obtained nogood. Constraint relaxation is an important element in improving efficiency of problem solving. The following subsections address these issues separately. First, we introduce a mechanism for saving nogoods when a relaxation occurs.

The technique we propose consists of explaining inferences with references to constraints (CR). A related technique has been used for dynamic domain splitting for numeric CSPs [14]. To enhance privacy support, the CRs do not necessarily stand one to one for a given constraint, but provide a way to signal when due to relaxations, a nogood is invalidated. Therefore, either some constraint, or whole sets of constraints, or both, can be associated with a CR each.

Example 1. The CSP of A_i, defined by the constraints $\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5$, can be associated with CRs from the set $(C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8)$ in the next way:

$$\begin{array}{cc} \gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5 & \{C_1\} \\ \gamma_1 & \{C_2, C_3\} \\ \gamma_1, \gamma_3 & \{C_4\} \\ \gamma_2 & \{C_5\} \\ \gamma_5 & \{C_6, C_7, C_8\} \end{array}$$

Each agent A_i is associated with a predefined CR, CR(A_i), (e.g. in the previous example is C_1). At any inference, the nogood resulting from the inference is tagged with the union between:

- CR(A_i), where A_i is the agent making the inference,
- CRs of the constraints used for inference, and
- the union of CRs tagging the nogoods used for inferences.

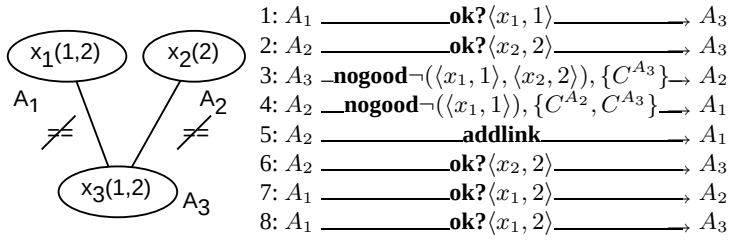

Fig. 4. Example of AFC$_c$ with maintenance of CRs.

In Figure 4 an example of AFC$_c$ with maintenance of CRs is given. CR(A_i) is denoted by C^{A_i}. The example uses the simple problem exploited in [30] to illustrate ABT.

when received *(***announce**,A_j,$\langle(t,i)\rangle$*)* **do**
  if *(interested in t)* **then**
    send **addlink** to all higher priority owners(t);
  if *(owner(t) and higher priority than A_j)* **then**
    add A_j in outgoing-list((t,i));
  send **domain**(local CRs for relaxed unary constraints);
when received*(***domain**,*crs)* **do**
  discard all nogoods in GCSP$_c^{A_i}$ that are tagged with a cr from crs;
  add crs to set of invalid CRs;
  check_agent_view.

Algorithm 2: Procedures of A_i for handling relaxation in AFC$_c$-GCSP.

The agent A_3 enforces the constraints $x_1{\neq}x_3, x_2{\neq}x_3, x_3{\in}\{1,2\}$. As in [24] each of these constraints can be represented with distinct CRs: $\{C_1^{A_3}, C_2^{A_3}, C_3^{A_3}\}$. For privacy reasons, A_3 can use more than three CRs, having several CRs for the same constraint. The separate relaxation of some of these constraints can be announced by broadcasting a set of CRs representing them. Therefore, each agent discards the nogoods tagged by CRs of relaxed constraints.

For simplicity, each agent in the example of Figure 4 uses only one CR. The agent A_3 infers the nogood $\neg(\langle x_1,1,1\rangle, \langle x_2,2,1\rangle)$ and tags it with C^{A_3}, (CR(A_3)). When the agent A_2 infers $\neg(\langle x_1,1,1\rangle)$, it tags this nogood with $\{C^{A_2}, C^{A_3}\}$ by adding C^{A_2} due to its constraint: $x_2{\in}\{2\}$.

We treat the extension of the domains of the variables as a constraint relaxation [24]. Namely, each time that a unary constraint restricting a variable is relaxed with the addition of some values, all the nogoods inferred due to the disappearing constraints have to be removed. A mechanism is needed to announce the relaxation and its parameters (added variables and invalidated CRs) to other agents.
For this reason we introduce the next features for algorithm extensions:

- **announce** message broadcasts a tuple (x,t,i), where x is a newly created variable of type t and with index i to all other agents. The receiving agents determine their interest in variable x and react depending on their interest and priority in one of the following ways (a) send an **addlink** message transporting the variable set $\{x\}$ (b) add the sending agent to its outgoing links or (c) discard the message.
- **domain** message broadcasts a set cr of obsolete constraint references. Any receiving agent removes all the nogoods having attached to them a constraint reference $cr \in crs$. The receiver of the message calls then the function *check_agent_view()*, making sure that it has a consistent proposal or that it generates nogoods.
- **nogood** messages transport *generic nogoods* $\neg N$ that contain assignments for meta-variable instances. These messages are multicasted to all agents interested in $\neg N$.[11] An agent A_i is interested in a generic nogood $\neg N$ if it has *interest in* any meta-variable in $\neg N$.

[11] The algorithm remains correct and terminates even if the nogoods are sent only to the target decided as in ABT.

– When an agent needs to revoke the creation of a new variable due to backtracking in his local solving algorithm, it assigns it a specific value from its domain indicating the deactivation of the variable and communicates it via an **ok?** message to all interested agents.

In order to avoid too many messages a broker agent can be introduced that maintains a static list of agents and their interest in variables of specific types comparable to a *yellow pages* service. In this case the agent that created a new variables only needs to request the broker agent for a list of interested agents and does not need to broadcast an **announce** message to all agents.

Theorem 3. *Whenever an existing extension of ABT is extended with the previous messages and is applied to DisGCSPs, the obtained protocols are correct, complete and terminate.*

Proof: Let us consider that we extend a protocol called P.
Completeness: All the generated information results by inference. If failure is inferred (when no new component is available), then indeed no solution exists.
Termination: Without introducing new variables, the algorithm terminates. Since the number of variables that can be generated is finite, termination is ensured.
Correctness: The resulting overall protocol is an instance of P, where the delays of the system agent initializing the search equals the time needed to insert all the variables generated before termination. Therefore the result satisfies all the agents and the solution is correct.

As described in procedure **backtrack** of Algorithm 3, generic nogoods inferred during AFC$_c$-GCSP can be sent either only to the lowest priority agent that built an assignment used in inferring the nogood, or to all agents knowing types involved in that nogood.

In procedure **check-agent-view** of Algorithm 3, an agent first checks:

– whether a relaxation is needed, namely an empty nogood tagged with a non-empty CRs is known.
– whether a relaxation is decided. A relaxation is typically decided when it is needed and a relaxation can be done by the current agent in order to invalidate a CR tagging an empty nogood.

Whenever a relaxation is decided, a counter variable is incremented, implying that generic constraints bounding domains are relaxed to extend those domains. The new problem is announced with **domain** and **announce** messages, as described in Algorithm 2. Each agent stores the set of invalidated CRs it knows so far.

5.3 Discussion

In Algorithm 3, links are established between two agents only once for any given type. One could actually establish a separate link for a subset of indices of variables of the given type, as detailed in the first example of this paper. There are two main reasons for reducing the generality of a link:

when received *(***ok?**,$\langle x_j, d_j \rangle$*)* **do**
- add $\langle x_j, d_j \rangle$ to *agent view*;
- **check_agent_view**;

when received *(***nogood**,A_j,$\neg N$,*CRs)* **do**
- **when** $\langle x_k, d_k \rangle$*, where* x_k *is not connected, is contained in* $\neg N$
  - send **addlink** to owner(x_k);
  - add $\langle x_k, d_k \rangle$ to *agent view*;
- **when** $\langle (t_k, i), d_k \rangle$*, where* t_k *is not connected, is contained in* $\neg N$
  - send **addlink** to all owners(t_k);
  - add $\langle (t_k, i), d_k \rangle$ to *agent view*;
- When CRs are valid, put $\neg N$,CRs in $GCSP_c^{A_i}$;
- **when** *have variable*
  - *old_value* $\leftarrow$ *current_value*;
- **check_agent_view**;
- **when** *(have variable) and (*old_value $=$ current_value*)*
  - send (**ok?**,$\langle x_i,$ *current_value*$\rangle$) to A_j;

procedure check_agent_view do
- if a relaxation is needed and decided, send corresponding **domain** and **announce**;
- **if** *have variable* **then**
  - **when** agent view *and* current_value *are not consistent*
    - **if** *no value in* D_i *is consistent with* agent view **then**
      - **backtrack**;
    - **else**
      - select $d \in D_i$ where *agent view* and d are consistent;
      - *current_value* $\leftarrow d$;
      - send (**ok?**,$\langle x_i, d \rangle$) to lower priority agents in *outgoing links*;
- **else**
  - **when** agent view *is not consistent*
    - **backtrack;**

procedure backtrack do
- (*nogoods*,CRss) $\leftarrow \{V \mid V =$ abstracted inconsistent subset of *agent view*$\}$;
- **when** *an empty set (*$\{\}$*,crs) is an element of (*nogoods,*CRss)*
  - broadcast (**nogood**,A_i,($\{\}$,crs));
  - **if** crs is empty **then** terminate this algorithm;
- **for every** $(V, cr) \in$ *(*nogoods,*CRss)* **do**
  - select $\langle x_j, d_j \rangle$ where x_j has the lowest priority in V;
  - send (**nogood**,A_i,(V, cr)) to owner(x_j);
  - **for every** $M^t \in V$ **do** send (**nogood**,A_i,(V, cr)) to owners(t);
  - remove $\langle x_j, d_j \rangle$ from *agent view*;
- **check_agent_view**;

Algorithm 3: Procedures of A_i for receiving messages in AFC$_c$-GCSP.

– While we prefer to reduce the overall network load, coarse-grained links lead to the generation of useless **ok?** messages.
– For privacy reasons it is preferable, if possible, not to reveal to all participants all values that were acceptable and proposed by the agents.

On the other side, too fine-grained links lead to:

– continuous rediscovery of the links, when links can be forgotten [23], and
– excessive delay in the establishment of the links and in getting updated validity information [3, 23, 4].

The optimal trade-off varies with the required properties of the algorithm and with the given problem. Strong privacy requests will encourage the use of very fine grained links.
The AFC_c algorithms allow agents to only deal with at most one variable at once. This was specified already in [30] as sufficient for modeling problems with more than one variable per agent. Then, multiple variables can be addressed introducing a separate 'virtual' (aka 'abstract') agent for each variable. As shown in the implementation in [32], a single name and address can be used for all abstract agents of a physical agent, as the incoming information can be easily dispatched locally according to the needs of the algorithm. This way, most common structures such as agent-view, nogood-stores, and outgoing-links can be shared. Agent-views have however to be implemented in a way allowing the assignments proposed by oneself to be also known, and assignments for higher priority variables, from the same physical agent, to be filtered out in *check-agent-view*.
An interesting phenomena appears in DisGCSPs with the generation of new variables for which an agent starts to propose messages. Whenever an agent starts to propose values for a new variable, a new abstract agent is created to deal locally with that variable. The creation of a new abstract agent consists actually only in the run of the procedures for receiving messages an additional time for the given new variable. All the structures are shared with the other agents. With fine-grained links, separate outgoing-links are used.

6 Evaluation

In order to test the applicability of our approach we implemented a prototype framework for Distributed Generative Constraint Satisfaction on top of ILOG's *JConfigurator* [12]. According to the GCSP approach, the user of this library defines the problem in terms of components, ports, and attributes and states generic constraints that apply to the set of all instances of a specific component type [12, 16].
Our framework provides a simple, experimental infrastructure for distributed reasoning among an arbitrary number of agents, each one capable of solving a local GCSP, whereby there are no limitations on the number of component types or the complexity of the constraints for the local GCSPs. Limitations for our tests were that ports are only allowed to connect the sub-systems with modules and that each component type is characterized solely by one integer attribute (with a finite numerical domain). However,

the number of ports of the sub-systems and the component instances were only limited to a theoretical maximum amount.

In order to be able to compare our DisGCSP framework with the conventional DisCSP framework, the example configuration problems were also modelled as a static CSP with all possible component instances already generated from the beginning, where their domain was extended with an additional value indicating their inactivity. So, we were able to compare the effect of defining nogoods and constraints *generically* on the number of interaction cycles between agents vs. classical constraint and nogood formulation in DisCSPs. In addition we found that additional computational costs for deriving minimal conflicts pays off given the potentially high communication overhead of message passing associated with additional interaction cycles.

Architecture. The core of our framework forms an *Agent* class that manages the *agent view*, and implements the methods for AFC$_c$, a variant of ABT algorithm in [30] (e.g., sending and processing **ok?** and **nogood** messages). Concrete agent instances (with their local problems) are implemented by subclassing and overriding the application specific methods for e.g., definition of components and constraints. Communication among agents is based on message passing via a *mediating agent* that provides capabilities for agent registration and system initialization and is capable of detecting when the distributed system reaches a stable state, i.e., a solution is found.

Conflict exchange. Conflict exchange among agents is based on serialization of the conflict information and automated (re-)construction of the generic constraint at the receiving agent's side.

Algorithm. Given the results from previous sections, several distributed constraint satisfaction algorithms can be employed to solve the distributed configuration problem. In our framework we currently employ AFC$_c$, a variant of sound and complete ABT algorithm in [30] with static ordering. This choice was mainly driven by the characteristics of the configuration domain, where the order of agents is highly determined by the supply chain setting (CAWICOMS project[12]). The extensions include the handling of multiple variables by aggregating variables according to their types: Agents can request links to variable *types* (component types in configuration terminology); thus, **ok?** messages contain the assignments of all currently existing variables of a given type, whereby each variable type is owned by exactly one agent. The computation of local solutions is performed by the underlying constraint solver. The application of dynamic agent ordering or configuration-specific heuristics is part of our future work.

Measurements. We performed some initial measurements (Table 1) of our framework for configuration problems of the structure described above varying the size of the configuration problem, the number of agents as well as the local search strategies and problem complexity in order to obtain significant distributed search and backtracking activity[13]. In the results shown in Table 1 we compare the behavior of the distributed systems, whereby we encoded the same problem both as *Generative* Constraint Satisfaction Problem with a given upper bound of possible component instances and as

[12] See http://www.cawicoms.org for reference.

[13] Note, that in the configuration domain, the number of co-operating agents of the companies involved in the supply chain is typically very low (< 10), the agents are usually loosely coupled and the problems are typically underconstrained.

static CSP. For the distributed reasoning process the identical variant of ABT search [30] (with support for multiple variables per agent) is employed, whereby in the case of the GCSP problem *generic* nogoods are exchanged among the agents according to AFC$_c$-GCSP. In both settings the explanation facilities for computing minimal nogoods were utilized.

It is well known that formalisms that extend the static CSP paradigm like Dynamic CSP or Generative CSP have its advantages for non-distributed problem solving both from the modeling, knowledge acquisition, and maintenance perspective as well as from the solution search point of view. In a distributed settings where the configuration constraints are distributed among several cooperating agents, the non-generic approach suffers from the problem of heavy message traffic that is induced by the increased number of required interaction cycles for finding the solution: While in the CSP encoding a receiving agent is only capable of computing minimal conflicts involving concrete variable instances, in the generic case the agent can deduce and report *generic* nogoods to the sending agent. It is the fact that - in the GCSP and configuration problem setting - the individual variable (i.e., component) instances of a given type are interchangeable which allows us to report a nogood that prevents the sending agent from communicating an interchangeable solution which would again cause an inconsistency for the receiving agent.

Figure 6 contains the time measurements for finding the first solution in five different

	AFC_c					$AFC_c\ GCSP$				
	(1)	(2)	(3)	(4a)	(4b)	(1)	(2)	(3)	(4a)	(4b)
Number of agents	3	6	10	12	12	3	6	10	12	12
Number of constraints	10	22	30	36	36	10	22	30	36	36
Number of component instances	30	120	140	164	164	30	120	140	164	164
Number of shared instances	12	27	65	87	87	12	27	65	87	87
Check-Time per agent	14,3	14,3	34,5	38,3	128	1,7	3,2	5,1	5	8,3
Overall number of recorded nogoods	165	211	594	705	3646	75	126	205	238	588
Overall number of Messages	477	644	1792	2635	16476	75	126	205	238	588

Fig. 5. Algorithm comparison

configuration scenarios with varying complexity, whereby the problem instances 4(a) and 4(b) are identical in terms of the problem size but differ in search complexity. The numbers are the average of several runs for each problem instance, differences occur due to the indeterministic behavior of the parallel execution of the agents. The problem sizes regarding the number of component types/agents are realistic for the scenarios addressed within the CAWICOMS project. Furthermore, the scenarios reflect the fact that in a supply chain setting only a smaller portion of the local configuration problems are shared among the agents. The local GCSPs are underconstrained; using more complex problems will result in an increase of the time needed for consistency checks and local solution search which is done by the constraint solver. The actual number of problem variables is determined by the number of component instances,

the cardinality variables for all types and the internally generated variables that allow the formulation of n-ary constraints. While the net search times are secondary[14], the experiments showed that the generative CSP performs significantly better in terms of required interaction cycles, stored nogoods and messages.

While message passing is quite cheap in our multi-threaded prototype, the costs of agent communication in real distributed environments are a crucial factor. The memory costs of storing nogoods are not a difficulty for our problem size due to the minimality of the nogoods and a compact representation. However, minimizing the number of search cycles by reducing the search space through the elimination of interchangeable solutions leads to overall performance enhancements especially in cases where the local configuration problems are complex. Beside the advantages of faster distributed solution search, the GCSP approach has significant advantages for the domain of real-world distributed configuration problems in terms of knowledge maintenance: the problem of modeling and maintaining shared agent knowledge and agent interdependencies is neglected in many DisCSP approaches and alleviated in a GCSP setting through the introduction of variable types and generic constraints, thus eliminating the need for error-prone problem encoding with static CSPs.
Finally, the experiments showed that the integration of distributed configuration capabilities into a commercial configuration tool like *JConfigurator* is feasible and lays the path for the application of distributed constraint solving to real-world environments.

7 Conclusions

Building on the definition of a centralized configuration task from [28], we formally defined a new class of CSP, termed generative CSP (GCSP), that generalizes the approaches of constraint-based configurator applications in use [8, 16]. The innovative aspects include an additional level of abstraction for constraints and nogoods. Constraints and nogoods can refer to types of variables. Furthermore, we extended GCSP to a distributed scenario, where this abstraction adapts well DisCSP frameworks for dynamic configuration problems (but it can be used in static models as well). We have described how this enhancement can be naturally integrated in a large family of existing asynchronous algorithms for DisCSPs and gave evidence for its practical application by evaluating it for typical distributed configuration problems.

References

1. A. Armstrong and E. F. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proc. of the 15th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, Nagoya, Japan, 1997.
2. V.E. Barker, D.E. O'Connor, J.D. Bachant, and E. Soloway. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, 32(3):298–318, 1989.

[14] The time measurements where made on a standard PC whereby the parallel agent threads run in one single process; overall memory consumption was in all DisGCSP test cases below 25 MB.

3. C. Bessière, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In *Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, page 772, Paphos, Cyprus, 2001.
4. C. Bessière, A. Maestre, and P. Meseguer. The abt family. In *Proc of JFNP*, 2002.
5. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. on Computing Systems*, 3(1):63–75, 1985.
6. R. Dechter and A. Dechter. Belief Maintenance in Dynamic Constraint Networks. In *Proc. 7th National Conf. on Artificial Intelligence (AAAI)*, pages 37–42, St. Paul, MN, 1988.
7. Boi Faltings and Santiago Macho-Gonzalez. Open constraint satisfaction. In *Proc. of CP'02*, 2002.
8. 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.
9. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *CP'02*, pages 462–476, Ithaca, September 2002.
10. A. Haselböck. *Knowledge-based configuration and advanced constraint technologies*. PhD thesis, Technische Universität Wien, 1993.
11. W. Havens. Nogood caching for multiagent backtrack search. In *Proc. of 14th National Conf. on Artificial Intelligence (AAAI), Agents Workshop*, Providence, Rhode Island, 1997.
12. U. Junker. Preference-based programming for Configuration. In *Proc. of IJCAI'01 Workshop on Configuration*, Seattle, WA, 2001.
13. U. Junker. QuickXPlain: Conflict Detection for Arbitrary Constraint Propagation Algorithms. In *Proc. of IJCAI'01 Workshop on Modelling and Solving problems with constraint*, Seattle, WA, 2001.
14. N. Jussien and O. Lhomme. Dynamic domain splitting for numeric CSP. In *European Conference on Artificial Intelligence*, pages 224–228, 1998.
15. Q.Y. Luo, P.G. Hendry, and J.T. Buchanan. Strategies for distributed constraint satisfaction problems. In *Proc. of the 13th International Workshop on DAI*, pages 207–221, 1994.
16. D. Mailharro. A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):383–397, 1998.
17. D.L. McGuiness and J.R. Wright. Conceptual Modeling for Configuration: A Description Logic-based Approach. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):333–344, 1998.
18. P. Meseguer and M. A. Jiménez. Distributed forward checking. In *CP DCS Workshop*, 2000.
19. S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proc. of 8th National Conf. on Artificial Intelligence (AAAI)*, pages 25–32, Boston, MA, 1990.
20. D. Sabin and E.C. Freuder. Configuration as Composite Constraint Satisfaction. In *Proc. of AAAI Fall Symposium on Configuration*, Cambridge, MA, 1996. AAAI Press.
21. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of 17th National Conf. on Artificial Intelligence (AAAI)*, pages 917–922, Austin, TX, 2000.
22. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. ABT with asynchronous reordering. In *Proc. of Intelligent Agent Technology (IAT)*, pages 54–63, Maebashi, Japan, October 2001.
23. M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Polynomial space and complete multiply asynchronous search with abstractions. In *IJCAI-01 DCR Workshop*, pages 17–32, Seattle, August 2001.
24. M.-C. Silaghi, D. Sam-Haroud, and B.V. Faltings. Maintaining hierarchically distributed consistency. In *Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP), DCS Workshop*, pages 15–24, Singapore, 2000.

25. M.-C. Silaghi, D. Sam-Haroud, and B.V. Faltings. Consistency maintenance for ABT. In *Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 271–285, Paphos, Cyprus, 2001.
26. Marius-Călin Silaghi. *Asynchronously Solving Distributed Problems with Privacy Requirements*. 2601, Swiss Federal Institute of Technology (EPFL), CH-1015 Ecublens, June 27, 2002.
27. M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2), 1997.
28. M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):307–320, 1998.
29. M. Yokoo. Asynchronous weak-commitment search for solving large-scale distributed constraint satisfaction problems. In *Proc. of 1st Int. Conf. on Multi-Agent Sytstems (ICMAS)*, pages 467–318, San Francisco, CA, 1995.
30. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proc. of 12th Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 614–621, Yokohama, Japan, 1992.
31. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Knowledge and Data Engineering*, 10(5):673–685, 1998.
32. M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proc. of the 3rd Int. Conf. on Multi-Agent Systems (ICMAS)*, pages 372–379, Paris, France, 1998.