# A Distributed Generative CSP Framework for Multi-Site Product Configuration

M. Zanker[1], D. Jannach[2], M.C. Silaghi[3] and G. Friedrich[1]

[1)] University Klagenfurt, Austria
e-mail:{markus.zanker,gerhard.friedrich}@uni-klu.ac.at
[2)] Technical University Dortmund, Germany
e-mail:dietmar.jannach@cs.uni-dortmund.de
[3)] Florida Institute of Technology (FIT), Melbourne, US
e-mail: Marius.Silaghi@fit.edu

**Abstract.** Today's configuration systems are centralized and do not allow manufacturers to collaborate online for offer-generation or sales-configuration activities. However, the integration of configurable products into the supply-chain of a business requires the cooperation of the various manufacturers' configuration systems to jointly offer valuable solutions to customers. As a consequence, there is a need for methods that enable independent specialized agents to compute such configurations. Several approaches to *centralized* configuration are based on constraint satisfaction problem (CSP) solving. Most of them extend traditional CSP approaches in order to comply to the specific expressivity and dynamism requirements of 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 extensible domains of problem variables. It also builds on the distributed CSP (DisCSP) framework, supporting configuration tasks where knowledge is distributed over a set of agents. Notably, the notions of constraint and nogood are further generalized, adding an additional level of abstraction and extending inferences to types of variables. An example application of the new framework describes modifications to the ABT algorithms and furthermore our evaluation indicates that the DisGCSP framework is superior to classic DisCSP for typical configuration task problem encoding.

## 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 that should be included in the configured product and can determine the physical component structure of the personalized product variant. Typically, there are technical and marketing restrictions on the valid parameter constellations and the physical layout. This has led manufacturers to develop methods for checking the feasibility of user requirements and for computing consistent solutions. Typically, this functionality is provided by product configuration systems (configurators), which have proven to be a successful application area for different AI techniques [18] such as description logics [11], or rule-based [2] and constraint-based solving algorithms. [5] describes the industrial use of constraint techniques for the configuration of large and complex systems

such as telecommunication switches and [10] is an example of a powerful commercialised tool based on constraint satisfaction.

However, companies find themselves having to cooperate with other highly specialized solution providers as part of a dynamic coalition to offer customized solutions. The level of integration present in today's digital markets implies that software systems supporting selling and configuration tasks may no longer be 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 a configuration task. This paper abstracts the *centralized* definition of a configuration task in [19] 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 extend DisCSPs to handle distributed configuration tasks by integrating the innovative aspects of local generative CSPs:

1. The constraints (and nogoods) are generalized such that they depend on the types rather than on the identities of variables. This also enables the following aspects to be treated more elegantly.
2. The number of variables of certain types that are active in the local CSP of an agent may vary depending on the state of the search process and is hence dynamic. In the DisCSP framework, the external variables existing within the system are predetermined.
3. The domain of the variables may vary dynamically. Some variables model possible connections and depend on the existence of components that could later be connected.

Importantly, we also describe the impact of the previously mentioned changes on asynchronous algorithms. In the following we motivate our approach with an example, Section 3 defines a generative CSP and in Section 4 a distributed generative CSP is formalized and presented together with extensions to current DisCSP frameworks. Finally, Section 5 evaluates DisGCSP encoding against classic DisCSP problem representation for typical configuration problems.

## 2 Motivating example

The following presents a typical example problem from the domain of product configuration ([5]) where interconnected systems support plug-in modules and problem specific constraints describe the legal combinations of module types and their capacity as well as their associated parameters. Figure 1 depicts a problem where systems consist of modules of different types, namely A-, B-, and C-modules, and have optional connection points for these modules (denoted as ports). For reasons of presentation the example focuses only on a small subset of a larger configuration problem of a technical system (see dotted lines). System 1 consists of A- and B-modules where system 2 may have only A- and C-modules plugged in. The A-modules also act as an interface between the two systems, i.e. they are shared by them. In addition, a module of any type can be either set as active or inactive. The initial situation in Figure 1 depicts the customer specific requirement that the configuration result contains at least one A-module that is connected via a port to the sub-system. According to the compatibility

restrictions that will be described in the following, the found solution includes two additional modules of type B and C, where all A- and C modules are set to active and all B-modules to inactive. The distribution aspect is inherent in this scenario, as the

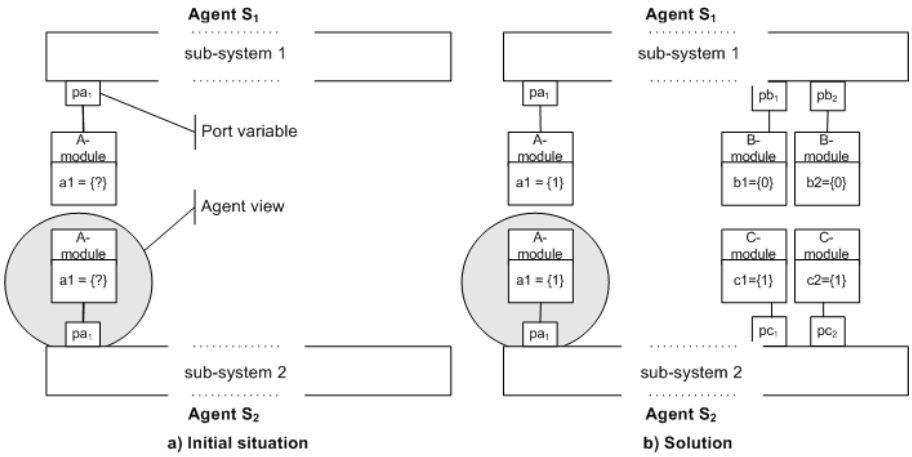


**Fig. 1.** Example problem

overall solution consists of sub-systems that are to be configured by different agents. We formalize this configuration problem as a CSP, where each port and each module is represented by a variable[1]. Since the exact number of problem variables is not known from the beginning, constraints cannot be directly formulated on concrete variables. Instead, comparable to programming languages, variable types exist that allow to associate a newly created variable with a domain and we can specify relationships in terms of *generic constraints*. [19] define a generic constraint $\gamma$ as a constraint schema, where meta-variables $M_i$ act as placeholders for concrete variables of a specific type $t$, denoted by the predicate $type(M_i) = t$. The subscript $i$ allows to distinguish between different meta-variables in one constraint[2]. In our example seven different types of problem variables exist, representing the ports for the three different module types ($t_{pa}, t_{pb}, t_{pc}$) and the activation status for each type of the modules ($t_a, t_b, t_c$) as well as a type ($t_{ct}$) of counter variables ($x_{type}$) for the number of instantiations of each type. The configuration constraints are distributed between the agents, i.e., each agent $S_i$ posesses a set of local constraints[3] $\Gamma^{S_i}$, i.e., $\Gamma^{S_1} = \{\gamma_1, \gamma_2, \gamma_5, \gamma_7, \gamma_9\}$ and $\Gamma^{S_2} = \{\gamma_3, \gamma_4, \gamma_6, \gamma_8, \gamma_{10}\}$, that are defined as follows:

*Agent $S_1$ ensures, that the amount of B-modules and its associated port variables must*

[1] Note, that the *sub-system* components themselves are not explicitly modeled, but only via their characterizing port variables.

[2] The exact semantics of generic constraints is given in Definition 2 in Section 3

[3] In the example we omit those constraints that ensure that once a port variable is assigned a value, the corresponding connected component variable must exist.

*not be above 3.*
$\gamma_1 : val(x_{pb}) \leq 3.$ and $\gamma_2 : val(x_b) \leq 3.$ where $val(x)$ is a predicate that gives the assigned value of variable $x$.
*Similarly for agent $S_2$, the amount of C-modules and its associated port variables must not be above 3.*
$\gamma_3 : val(x_{pc}) \leq 3.$ and $\gamma_4 : val(x_c) \leq 3.$
*Agent $S_1$ resp. $S_2$ check that there are more B- as well as C-modules than A-modules in a configured system.*
$\gamma_5 : val(x_b) > val(x_a).$ and $\gamma_6 : val(x_c) > val(x_a).$
*Agent $S_1$ resp. $S_2$ ensure that all B- resp. all C-modules have set the same activation status.*
$\gamma_7 : type(M_1) = t_b \wedge type(M_2) = t_b \wedge val(M_1) = val(M_2).$ and
$\gamma_8 : type(M_1) = t_c \wedge type(M_2) = t_c \wedge val(M_1) = val(M_2).$
*For agent $S_1$ A- and B-modules must not have the same activation status.*
$\gamma_9 : type(M_1) = t_a \wedge type(M_2) = t_b \wedge val(M_1) \neq val(M_2).$
*For agent $S_2$ A- and C-modules must have the same activation status.*
$\gamma_{10} : type(M_1) = t_a \wedge type(M_2) = t_c \wedge val(M_1) = val(M_2).$

During the search process the search space is continuously extended by the instantiation of additional problem variables, until a solution is found that satisfies all the constraints of each agent. The $Agent\ view$ contains the problem variables shared between agents in order to assure the evaluation of local constraints. In our case $\gamma_6$ and $\gamma_{10}$ are so-called inter-agent constraints, that require agent $S_2$ to have access to all A-modules and its associated port variables.
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 the local configurators and we detail extensions to DisCSP algorithms.

## 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 where problem variables representing components of the final system are *generated* dynamically as part of the solution process because their total number cannot be determined beforehand. The framework is called *generative* CSP (GCSP) [6, 19]. This kind of dynamicity extends the approach of dynamic CSP (DCSP) formalized by Mittal and Falkenhainer [12], where all possibly involved variables are known from the beginning. This is needed because the activation constraints reason on the variable's activity state. [13] 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 [4] where constraints can be added or removed independently of the initial problem statement. The dynamicity occuring in a GCSP differentiates from the one described in [4] in the sense that a GCSP is extended in order to find a consistent solution and the latter 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 configuration task specific formulation in [19] and applies to the wider range of synthesis problems.

**Definition 1 (Generative constraint satisfaction problem (GCSP)).** *A generative constraint satisfaction problem is a tuple GCSP($X$, $\Gamma$, $T$, $\Delta$), where:*

- *$X$ is the set of problem variables of the GCSP and $X_0 \subseteq X$ is the set of initially given variables.*
- *$\Gamma$ 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 $t_i \in T$ 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.*
- *$\Delta$ is a total relation on $X \times (T, N)$, where $N$ is the set of positive integer numbers. Each tuple $(x, (t, i))$ associates a variable $x \in X$ 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 $\Delta$ and returns the type $t \in T$ for $x$ and the function $index(x)$ returns the index of $x$.*

By generating additional variables, a previously unsolvable CSP can become solvable, which is explained by the existence of variables that hold the number of variables.

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 [19].

**Example** Given $t_a$ as the type of variables representing *A-modules* and $t_{pa}$ as the type of *port* variables that are allowed to connect to *A-modules*, then the domain of the *pa* variables $dom(t_{pa})$ must contain references to *A-modules*. This is specified by defining $dom(t_{pa}) = \{1, \ldots, ub\}$, where $ub$ is an upperbound on the number of variables of type $t_a$, and 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., $type(M_1) = t_{pa} \wedge val(M_1) \leq x_{t_a}$. With the help of the $index()$ function concrete variables can then be referenced.
Referring to our introductory example we can formalize the local GCSP of agent $S_1$ in the initial situation (see Figure 1) as $X^{S_1} = \{x_a, x_{pa}, x_b, x_{pb}, x_{ct}, a_1, pa_1\}$, $\Gamma^{S_1} = \{\gamma_1, \gamma_2, \gamma_5, \gamma_7, \gamma_9\}$, $T^{S_1} = \{t_{ct}, t_a, t_{pa}, t_b, t_{pb}\}$ and $\Delta^{S_1} = \{(x_a, (t_{ct}, 1)), (x_{pa}, (t_{ct}, 2)), (x_b, (t_{ct}, 3))(x_{pb}, (t_{ct}, 4)), (x_{ct}, (t_{ct}, 5)), (a_1, (t_a, 1))\ (pa_1, (t_{pa}, 1))\}$. The $index(S_1)$ function returns 1, which indicates that $a_1$ is the first A-module instance. The domains of variables are consequently defined as $dom(t_a) = dom(t_{pa}) = dom(t_b) = dom(t_{pb}) = dom(t_{ct}) = \{1, \ldots, ub\}$, where the domains for the port variables are additionally limited by domain constraints (e.g., $\gamma_1$).

**Definition 2 (Generic constraint).** *A generic constraint $\gamma \in \Gamma$ formulates a restriction on the meta-variables $M_a, \ldots, M_k$. 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* $type(x_j) = type(M_i)$*.*

Note, that generic constraints can also formulate restrictions on specific initial variables from $X_0$ by employing the $index()$ function.
Consider the GCSP($X$, $\Gamma$, $T$, $\Delta$) 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 $M_i$, then the consistency of generic constraints is defined as follows:

**Definition 3 (Consistency of generic constraints).** *Given an assignment tuple* $\theta$ *for the variables* $X$*, then* $\gamma$ *is said to be satisfied under* $\theta$*, iff* $\forall x_a, \ldots, x_k \in X$ : $type(x_a) = type(M_a) \wedge \ldots \wedge type(x_k) = type(M_k) \rightarrow \gamma[M_a|_{x_a}, \ldots, M_k|_{x_k}]$ *is satisfied unter* $\theta$*, 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, e.g., given a fragment of a GCSP of agent $S_1$ (excluding counter and port variables) with $X^{S_1} = \{a_1, b_1, b_2\}$, $T^{S_1} = \{t_a, t_b\}$ and $\Delta^{S_1} = \{(a_1, (t_a, 1)), (b_1, (t_b, 1)), (b_2, (t_b, 2))\}$, the satisfiability of the generic constraint $\gamma_9$ is checked by testing the following conditions: $val(a_1) \neq val(b_1)$, $val(a_1) \neq val(b_2)$.

**Definition 4 (Solution for a generative CSP).** *Given a generative constraint satisfaction problem GCSP(*$X_0$*,* $\Gamma$*,* $T$*,* $\Delta_0$*), then its solution encompasses the finding of a set of variables* $X$*, type and index assignments* $\Delta$ *and an assignment tuple* $\theta$ *for the variables in* $X$*, s.t.*

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

Note, that we do not impose a minimality criterium on the number of variables in our solution, because in practical applications different optimization criteria exist, such as total cost or flexibility of the solution, thus non-minimal solutions can be preferred over minimal ones.
The calculated solution (excluding counter variables) for the local GCSP of agent $a_1$ consists of $X^{S_1} = \{a_1, pa_1, b_1, b_2, pb_1, pb_2\}$, $\Delta^{S_1} = \{(a_1, (t_a, 1)), (pa_1, (t_{pa}, 1)), (b_1, (t_b, 1)), (b_2, (t_b, 2)), (pb_1, (t_{pb}, 1)), (pb_2, (t_{pb}, 2))\}$ and the assignment tuple $a_1 = 1$, $pa_1 = 1$, $b_1 = 0$, $b_2 = 0$, $pb_1 = 1$ and $pb_2 = 2$. Thus, $b_1, \ldots, b_2$ and $pb_1, \ldots, pb_2$ are the names of *generated* variables.
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. Our GCSP definition extends the definition from [19] 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 [19] only variables of a single type, i.e., component variables, can be created. Current CSP implementations of configuration systems (e.g., [10] [5]) 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 [19] 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 for the formulation of restrictions. Examples are the LCON language used in the COCOS project [19], or the configuration language of the ILOG Configurator [10].
Note, that the set of variables $X$ can 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 DisGCSP Framework

Algorithms for configuration applications need to guarantee 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) [21]. An enhanced version for several variables per agent is described in [22]. [3] shows how ABT can be adapted to networks where not all agents can directly communicate to one another. [7] 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 [20, 15, 17]. [14] 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.

We show how the basic DisCSP framework for ABT [21] can be applied to a scenario of distributed product configuration. Therefore, improving the performance of ABT with extensions as referenced above is straightforward. We summarize in the following the properties of the ABT algorithm that guarantee its correctness and completeness [21]. Then we apply this DisCSP framework to a scenario where each agent locally solves a generative constraint satisfaction task. Each time an agent extends the solution space of his local GCSP by creating an additional variable, the DisCSP setting is transformed into a new DisCSP setting, which again has all properties required by asynchronous search to correctly function.

### 4.1 Asynchronous Search

We summarize the characteristics of asynchronous search algorithms like ABT [21], reformulated to allow agents to know only the constraints that they enforce. They are considered as follows:

1. $A = \{S_1, \ldots S_n\}$ is a set of $n$ totally ordered agents (i.e. representing different sub-systems), where $S_i$ has priority over $S_j$ if $i < j$.
2. Each agent $S_i$ owns a variable[4] and knows all the constraints that involve its variable and only variables of higher priority agents.[5] The constraints known by $S_i$ are referred to as its local constraints, denoted $\Gamma^{S_i}$ and $S_i$ is *interested in* those variables that are contained in its local constraints. A *link* exists between two agents if they share a variable, that is directed from the agent with higher priority to the agent with lower priority. A link from agent $S_1$ to agent $S_2$ is referred to as an *outgoing link* of $S_1$ and an *incoming link* of $S_2$.
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 $S_i$ is a set of the most recent assignments received for those variables agent $S_i$ is interested in.
5. The agents communicate using the following types of messages, where channels without message loss are assumed:
   - **ok?** message. Agents with higher priorities communicate via each **ok?** message an assignment for their variable 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.

### 4.2 Framework for DisGCSP

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. As constraints employ meta-variables, the *interest* of an agent in variables needs to be redefined:

**Definition 5 (Interest in variables).** *An agent $S_j$ owning a local $GCSP^{S_j}(X^{S_j}, \Gamma^{S_j}, T^{S_j}, \Delta^{S_j})$ is said to be interested in a variable $x \in X^{S_h}$ of an agent $S_h$, if there exists a generic constraint $\gamma \in \Gamma^{S_j}$ formulating a restriction on the meta-variables $M_a, \ldots, M_k$, where $type(M_i) \in T^{S_j}$ is the defined variable type of the meta variable $M_i$, and $\exists M_i \in M_a, \ldots, M_k : type(x) = type(M_i)$.*

[4] As described later, one can see this variable as a tuple of variables treated simultaneously.

[5] In the original description of ABT, an agent also knows constraints on variables of higher priority agents.

**Definition 6 (Distributed generative CSP).** *A distributed generative constraint satisfaction problem has the following characteristics:*

- $A = \{S_1, \ldots, S_n\}$ *is a set of* $n$ *agents, where each agent* $S_i$ *owns a local* $GCSP^{S_i}$ *(*$X^{S_i}$*,* $\Gamma^{S_i}$*,* $T^{S_i}$*,* $\Delta^{S_i}$*).*
- *All variables in* $\bigcup_{i=1}^{n} X^{S_i}$ *and all type denominators in* $\bigcup_{i=1}^{n} T^{S_i}$ *share a common namespace, ensuring that a symbol denotes the same variable, resp. the same type, with every agent.*
- *For every pair of agents* $S_i, S_j \in A$ *and for every variable* $x \in X^{S_j}$*, where agent* $S_i$ *is interested in* $x$*, must hold* $x \in X^{S_i}$*.*
- *For every pair of agents* $S_i, S_j \in A$ *and for every shared variable* $x \in X^{S_i} \cap X^{S_j}$ *the same type and index must be associated to* $x$ *in the local GCSPs of the agents, i.e.,* $type^{S_i}(x) = type^{S_j}(x) \wedge index^{S_i}(x) = index^{S_j}(x)$*.*

Consequently, for every pair of agents $S_i, S_j \in A$ and for every shared variable $x \in X^{S_i} \cap X^{S_j}$ 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 7.** *Given a distributed generative constraint satisfaction problem among a set of* $n$ *agents then its solution encompasses the finding of a set of variables* $X = \bigcup_{i=1}^{n} X^{S_i}$*, type and index assignments* $\Delta = \bigcup_{i=1}^{n} \Delta^{S_i}$ *and an assignment tuple* $\theta = \bigcup_{i=1}^{n} \theta^{S_i}$ *for every variable in* $X$*, s.t. for all agents* $S_i : X^{S_i}$*,* $\Delta^{S_i}$ *and* $\theta^{S_i}$ *are a solution for the local* $GCSP^{S_i}$ *of agent* $S_i$*.*

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

**Definition 8 (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 9 (Generic nogood).** *A generic nogood takes the form* $\neg N$*, where* $N$ *is a set of generic assignments for distinct meta-variables.*

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 ($cr$) [16]. Any inference has to attach the $cr$s associated to arguments into the obtained nogood. We treat the extension of the domains of the variables as a constraint relaxation [16]. 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 $cr \in CR$. The receiver of the message calls then the function *check_agent_view()* detailed in [21], 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 $\neg N$.[6] An agent $S_i$ is interested in a generic nogood $\neg N$ if it has *interest in* any meta-variable in $\neg N$.
- When an agent needs to revoke the creation of a new variable due to backtracking in his local solving algorithm, he 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 1.** *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.

## 5 Evaluation

In order to test the applicability of our approach, we implemented a prototype for distributed generative constraint satisfaction on top of ILOG's *JConfigurator* [8]. *JConfigurator* is a Java library providing an API for modeling and solving configuration problems based on an underlying object-oriented constraint solver. Consistent with 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 [8, 10].
Our framework provides a simple, experimental infrastructure for distributed reasoning among an arbitrary number of agents, each of which is capable of solving a local

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

GCSP, where there are no limitations on the number of component types or the complexity of the constraints for the local GCSPs. However, for the purposes of evaluating the framework, despite the lack of benchmarking problems, we restricted the *structure* of the configuration problem to being similar to the example in Section 2 which still captures the main characteristics of configuration problems. Our tests were limited to ports that connected the sub-systems with the modules and component types that were characterized solely by one integer attribute (with a finite numerical domain). However, the maximum number of the sub-system ports and component instances were only limited to a theoretical value.

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

**Architecture.** The framework's core is an *Agent* class that manages the *agent view* and implements a variant of Yokoo's Asynchronous Backtracking algorithm (i.e., sending and processing **ok?** and **nogood** messages). Concrete agent instances (with their local problems) are implemented by subclassing and overriding application specific methods, for example, the 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 detection and exchange.** The computation of nogoods (minimal conflicts) in the case that the agent view is inconsistent, is based on Junker's QUICKXPLAIN algorithm [9], a efficient non-intrusive conflict-detector that recursively partitions the problem into subproblems of half the size and skips those that do not contain an element of the propagation-specific conflict[7]. In the current version, we only compute a single minimal, conflict in each backtracking step, future work will include the concurrent computation of several conflicts. A computed conflict contains information about the number of variables involved in the conflict as well as inconsistent variable assignments, where we can detect when the inconsistency arises solely from variable cardinalities which further improves the distributed search performance. Conflict exchange among agents is based on serialization of the conflict information and the receiving agent's automated (re-)construction of the generic constraint.

**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 a variant of Yokoo's sound and complete ABT algorithm without employing enhancements like dynamic agent ordering [1] or Aggre-

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

gation Search [14]. This choice was mainly driven by the characteristics of the configuration domain, where the order of the agents is mainly determined by the supply chain setting. 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, where each variable type is owned by exactly one agent. The computation of local solutions is performed by the underlying constraint solver. The additional task of applying dynamic agent ordering or configuration-specific heuristics remains part of our future work.

**Measurements.** Several initial tests (Table 1) were carried out on our framework using the configuration problems as 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[8]. The results shown in Table 1 present the behavior of the various distributed systems for the same problem ecoded both as *Generative* Constraint Satisfaction Problem with a given upper bound of possible component instances and as a static CSP. For distributed reasoning, the identical variant of Yokoo's ABT search (with support for multiple variables per agent) is employed, where in the case of the GCSP problem *generic* nogoods are exchanged among the agents. In both settings the explanation facilities for computing minimal nogoods were utilized.

It is well known that formalisms that extend the static CSP paradigm such as Dynamic CSP or Generative CSP have advantages for non-distributed problem solving both from modeling, knowledge acquisition, and maintenance perspectives as well as from a 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 messaging traffic that is induced by the increased number of required interaction cycles for finding a solution. In the case of traditional CSP encoding, a receiving agent is only capable of computing minimal conflicts involving concrete variable instances, however 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 - individual variable (i.e., component) instances of a given type are interchangeable. Therefore, reporting a nogood prevents the sending agent from communicating an interchangeable solution which would again cause an inconsistency for the receiving agent.

Table 1 contains the average time measurements for finding the first solution in five different configuration scenarios with varying complexity. Each problem instance was examined several times as differences occur due to the indeterministic behavior of the parallel execution of the agents. The problem sizes (i.e. the number of component types or 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 small portion of the local configuration problems are shared among the agents. The local GCSPs are underconstrained; using more complex problems would result in an increase in

[8] 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.

**Table 1.** Comparison of DisGCSP and DisCSP encoding

| Encoding | Nbr. of agents | Nbr. of CT | Nbr. of inst | Shared inst | Overall time | Check-time | NG | Msgs | Checks |
|---|---|---|---|---|---|---|---|---|---|
| **DisGCSP(1)** | 3 | 10 | 30 | 12 | 3.25 | 1.72 | **25** | **75** | 134 |
| **DisCSP(1)** | | | | | 23.20 | 14.30 | **165** | **477** | 820 |
| **DisGCSP(2)** | 6 | 22 | 120 | 27 | 8.53 | 3.21 | **40** | **126** | 210 |
| **DisCSP(2)** | | | | | 37.47 | 14.28 | **211** | **644** | 1105 |
| **DisGCSP(3)** | 10 | 30 | 140 | 65 | 13.90 | 5.05 | **63** | **205** | 375 |
| **DisCSP(3)** | | | | | 89.60 | 34.50 | **594** | **1792** | 3024 |
| **DisGCSP(4a)** | 12 | 36 | 164 | 87 | 15.32 | 5.03 | **66** | **238** | 403 |
| **DisCSP(4a)** | | | | | 127.44 | 38.30 | **705** | **2635** | 4094 |
| **DisGCSP(4b)** | 12 | 36 | 164 | 87 | 41.02 | 8.30 | **136** | **588** | 889 |
| **DisCSP(4b)** | | | | | 2600 | 128.00 | **3646** | **16476** | 23452 |

**CT**: component/variable types
**inst**: instances
**shared instances**: shared component instances
**Check-time**: consistency, search and explanation per agent
**NG**: overall number of recorded nogoods/backtracks
**Msgs**: overall number of messages
**Checks**: overall number of consistency checks and searches

the amount of time needed for consistency checks and the 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[9], the experiments showed that the generative variant performs significantly better in terms of required interaction cycles, stored nogoods and messages.

Figure 2 visualizes the run times for the different sample problems. Note that problem instances 4(a) and 4(b) are identical in terms of problem size and distribution among agents but differ in search complexity, i.e., problem 4(b) contains a problematic constraint constellation that causes the run-times of the static CSP approach to increase dramatically. While message passing is quite cheap in our multi-threaded prototype, the cost of agent communication in real distributed environments is a crucial factor. Memory requirements for storing nogoods are not problematic because they are minimal and can hence be represented in a compact way; 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

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

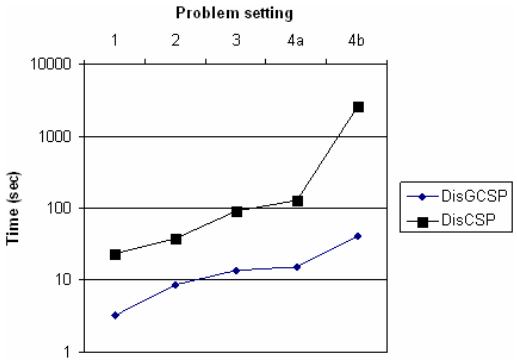


**Fig. 2.** Comparing run times for different problem settings.

local configuration problems are complex. Beside the advantage of offering a 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 DisGCSP setting through the introduction of variable types and generic constraints, thus eliminating the need for error-prone task of encoding problems as 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 foundation for the application of distributed constraint solving in real-world environments.

## 6 Conclusions

Building on the definition of a centralized configuration task from [19], we formally defined a new class of CSP, termed generative CSP (GCSP), that generalizes the approaches of current constraint-based configurator applications [5, 10]. The innovative aspects include an additional level of abstraction for constraints and nogoods. Constraints and nogoods may consist of variable types instead of solely variables. Furthermore, we extended GCSP to a distributed scenario, allowing DisCSP frameworks to be adapted to dynamic configuration problems (but it can be used in static models as well) and described how this enhancement can be integrated into a large family of existing asynchronous DisCSP algorithms. Initial evaluations indicate that GCSP is practical 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.

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. 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.
5. 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.
6. A. Haselböck. *Knowledge-based configuration and advanced constraint technologies*. PhD thesis, Technische Universität Wien, 1993.
7. W. Havens. Nogood caching for multiagent backtrack search. In *Proc. of 14th National Conf. on Artificial Intelligence (AAAI), Agents Workshop*, Providence, Rhode Island, 1997.
8. U. Junker. Preference-based programming for Configuration. In *Proc. of IJCAI'01 Workshop on Configuration*, Seattle, WA, 2001.
9. 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.
10. D. Mailharro. A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):383–397, 1998.
11. 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.
12. 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.
13. D. Sabin and E.C. Freuder. Configuration as Composite Constraint Satisfaction. In *Proc. of AAAI Fall Symposium on Configuration*, Cambridge, MA, 1996. AAAI Press.
14. 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.
15. 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.
16. 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.
17. 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.
18. M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2), June, 1997.
19. M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):307–320, 1998.
20. 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.
21. 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.
22. 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.