Modeling and Solving Distributed Configuration Problems: A CSP-Based Approach

Dietmar Jannach and Markus Zanker

Abstract—Product configuration can be defined as the task of tailoring a product according to the specific needs of a customer. Due to the inherent complexity of this task, which for example includes the consideration of complex constraints or the automatic completion of partial configurations, various Artificial Intelligence techniques have been explored in the last decades to tackle such configuration problems. Most of the existing approaches adopt a single-site, centralized approach. In modern supply chain settings, however, the components of a customizable product may themselves be configurable, thus requiring a multisite, distributed approach. In this paper, we analyze the challenges of modeling and solving such distributed configuration problems and propose an approach based on Distributed Constraint Satisfaction. In particular, we advocate the use of Generative Constraint Satisfaction for knowledge modeling and show in an experimental evaluation that the use of generic constraints is particularly advantageous also in the distributed problem solving phase.

Index Terms—Product configuration, distributed constraint satisfaction

1 Introduction

Product configuration systems are software applications that help the technical engineer, sales representative or the end customer to tailor a customizable product according to his or her specific needs. From a business perspective, such systems are a central element in any mass-customization business strategy, which aims to deliver individualized products at the costs of mass-production [38]. Usually, configurable products are assembled from predefined, parameterizable and interconnectable components where the set of legal parameter constellations and component connections is bound by a set of technical or marketing-related constraints. A typical product configuration system (configurator) therefore supports tasks such as checking whether a given configuration is consistent with the defined constraints, the automated completion of partial configurations, pricing or the generation of a bill-of-materials.

In general, building configuration systems is considered challenging both with respect to domain modeling and configuration problem solving. Regarding the modeling aspect, the main problem consists of capturing the complex and in some domains frequently changing knowledge, e.g., about which components can be combined or connected with each other. From the problem solving perspective, checking the consistency of a configuration or automatically completing a configuration can also be nontrivial due to the computational complexity of the task.

Over the last decades, several techniques from the field Artificial Intelligence have therefore been applied to product configuration problems, starting with early 1980s rule-based techniques [34], [6], over logic-based [37] and resource-based approaches [29] to more recent constraint-based solutions [12], [33] and preference programming [30].

One main assumption of most approaches to building product configurators is that there exists one single knowledge base containing—among other pieces of information—all the component descriptions and configuration constraints. However, increased levels of supply chain integration, a higher degree of specialization of suppliers as well as advanced interfirm modularity [50] or recent multimarket package procurement models [7] raised the demand for noncentralized solutions.

A typical example for such a multimarket package procurement problem is the configuration of telecommunication switches as described in [2]. In this scenario, the final product for the customer—a large-scale telephone switching system—consists of a configurable main switch but also of subcomponents, which are provided by different suppliers and are themselves configurable. The suppliers of the subcomponents are interested in keeping their detailed configuration and pricing rules private for business reasons. This makes the design of a centralized solution infeasible. In such distributed configuration scenarios, additional issues with respect to modeling and problem solving arise. In the context of the modeling task, the companies participating in the joint configuration process for example have to share and integrate parts of their configuration knowledge. Typically, some parts of the *configuration models* consisting, e.g., of component descriptions and part-of hierarchies, will be public to others, while some other pieces of knowledge are not shared for the above-mentioned reasons. Thus, some common form of modeling or knowledge integration and alignment is required.

Similarly, also new challenges arise for the problem solving phase. Often, the configurators in supply networks are organized in some hierarchical or sequential manner [11],

- *D. Jannach is with the Department of Computer Science, TU Dortmund, Dortmund 44221, Germany. E-mail: dietmar.jannach@tu-dortmund.de.*
- *M. Zanker is with the Institute of Applied Informatics, Alpen-Adria Universität Klagenfurt, Universitätsstraße 65-67, Klagenfurt 9020, Austria. E-mail: markus@ifit.uni-klu.ac.at.*

Manuscript received 23 June 2011; revised 21 Oct. 2011; accepted 31 Oct. 2011; published online 14 Nov. 2011.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2011-06-0369. Digital Object Identifier no. 10.1109/TKDE.2011.236.

[2]. Finding a solution to the overall configuration problem therefore requires not only the definition of communication and data exchange protocols but also advanced reasoning techniques. An example would be the capability for distributed backtracking in constraint-based approaches.

In this paper, we will first review requirements and challenges of modeling and solving distributed configuration problems. Next, we propose 1) a constraint-based framework for modeling and solving distributed configuration problems, which relies on Generative Constraint Satisfaction in order to support an intuitive modeling process and 2) a corresponding extension of an existing Distributed Constraint Satisfaction algorithm for the problem solving phase. Afterward, the results of a comparative evaluation on a real-world-inspired configuration problem are presented, which show the advantages of our method when compared with a standard constraint-based approach. The paper ends with a summary of our contributions and an outlook on future work.

2 Modeling (Distributed) Configuration Problems

2.1 From Rule-Based Systems to Component-Oriented Approaches

The choice of an appropriate mechanism for representing the domain knowledge is a crucial factor for the economic success of real-world configuration systems, in particular because configuration knowledge bases can become large and complex and are subject to frequent changes. For example, in the case of Digital's pioneering R1/XCON [34] rule-based configurator for VAX computer systems of the 1980s, the core configurator comprised more than 10,000 rules out of which 40 percent were changed each year. In addition, highly specialized teams of experts and engineers are required to maintain the knowledge base [6]. Such rule-based production systems, which are based, e.g., on the OPS5 language, were a popular knowledge representation mechanism for early expert systems but were soon replaced by other, declarative or logic-based formalisms in particular because of questions of maintainability or the problem of undesired side-effects that can arise when rules are changed.

With respect to the development of configuration knowledge bases, one of the key limitations of rule-based systems lies in the fact that the knowledge base structure does not reflect the structure of the configurable artifact itself. In reality, configurable products are typically assembled from a set of predefined components. The configuration knowledge therefore often consists of business-related or technical descriptions of how these components can be assembled, combined, parameterized, and interconnected to provide the functionality that is required by customers. Already in 1989, Mittal and Frayman therefore proposed a general model for the configuration task, which is based on the above mentioned concepts of components, connection points (ports), constraints and some other typical concepts such as *key components* and *functional architectures* [35]. Up to today, this general component-port model can be considered the most prevalent one in research and practice. Technically, the general model can be implemented in a variety of ways, e.g., based on (generative) constraint satisfaction [43], [41], [12], description logics [37], first-order logic [16], weighted constraint rules [44] or in an object-oriented and constraint-based variant [33].

From the conceptual viewpoint, Soininen et al. proposed a general ontology of configuration and a set of modeling concepts that are required in typical configuration knowledge bases [49]. One of their main goals was to synthesize previous works such as resource-based, structure-based or connection-based modeling approaches in order to develop an implementation-independent problem conceptualization that can be used to share and reuse configuration knowledge. A similar approach was proposed in [14] in which the Unified Modeling Language (UML) was used as a domain-specific language for configuration modeling. In their approach, the semantics of the graphical models were defined by providing a mapping to a representation of the knowledge in first-order logic.

2.2 Constraint-Based Approaches

Among the above-mentioned technical approaches to configuration problem solving, constraint-based techniques have always played an important role. One of the earliest constraint-based configurators was the COSSACK system [22], which was developed already in the mid-1980s. Configuration problems have also been a test bed for advanced constraint-based techniques, for example to analyze aspects of variable interchangeability and symmetry, soft constraints or questions of interactive problem solving, see, e.g., [10], [25], [4], or [3].

Classical Constraint Satisfaction Problems (CSPs) consist of a set of variables with finite domains and a set of constraints that describe allowed value assignments for variables. A solution to a CSP consists of an assignment of values to all the variables in a way that none of the constraints is violated. If we consider a typical car configuration problem [36], the possible engine types of a car could be encoded as a variable $engine = \{small, med, large\}$ and the available frame types as $frame = \{convertible, sedan, hatchBack\}$.

Such a representation is simple and intuitive at first glance. In the configuration domain, however, some decisions to be taken—for example on the type of the car's sunroof—are only relevant depending on a previous choice. In the car configuration problem, a sunroof might only be available when the luxury package was chosen first. In [36], Mittal and Frayman introduced *Dynamic Constraint Satisfaction Problems* (DCSPs), which are sometimes referred to as *Conditional CSPs*. In DCSPs, the knowledge base contains additional *activity constraints* that can be used to model under which circumstances a variable is part of the configuration problem. An example of such a *Require Variable* (RV) constraint could be: $package{=}luxury \Longrightarrow_{RV} sunroof$.

This constraint expresses that the variable *sunroof* only requires a value assignment when the variable *package* has the value *luxury*. Such activity constraints represent an intuitive way of encoding configuration knowledge, in particular because optional components are often found in product configuration problems. In [42], Soininen et al. showed that DCSPs are favorable when compared with CSP from the perspective of modularity of the knowledge base,

where modularity means that small changes in the configuration knowledge only lead to small changes in the CSP encoding. A detailed analysis of computational properties and the expressiveness of CSPs and these conditional CSPs is given in [23].

However, although DCSPs have advantages over CSPs, certain aspects of the configuration knowledge that are typically found in practice cannot be encoded compactly using this scheme. First, in many domains, multiple instances of the same component type can be part of a configuration. For example, when configuring large-scale telecommunication switches as described in [12], multiple modules of the same type can be mounted onto frames, each of which can be individually parameterized. In a DCSP encoding, each component instance and its attributes have to be modeled explicitly as individual (numbered) variables although they are structurally identical and exchangeable. Second, according to the above-mentioned component-port model, in some domains it is important to know how the individual parts of a configuration are interconnected. The concept of ports has therefore to be encoded in the (D)CSP scheme; the main modeling problem here is that the domain (the set of possible values) of a variable representing a port is either another port or a component. In other words, the allowed values for such a variable are determined by the number of actually existing components (or ports) in a given configuration.

Due to these limitations, a *generative constraint satisfaction formalism* [43] has been proposed as a means for modeling and solving configuration problems more intuitively. The GCSP addresses the above-mentioned problems and has been successfully used to develop real-world systems [12], [41] and software libraries such as the one described in [33].

Note that from a knowledge representation perspective, these GCSP-based systems often adopt a frame-based or object-oriented approach, where the central modeling element is a "component type." The configuration task in such generative approaches consists of two interrelated steps: 1) determining the number of instances for each component type and 2) assigning values to component attributes and ports, which are modeled as constraint variables.

A *Generative Constraint Satisfaction Problem* (GCSP) therefore has the following characteristics which cannot be found in standard CSPs. First, the number of components in a configuration (and consequently the number of variables) of the problem is not known in advance and components can be generated "on-demand" during the search process. Second, constraints are *generic*, that is, they are defined over variable types and apply to all instances of the same type. Finally, when connection points are modeled as variables, their domains must be variable or extensible, because the set of possible components to connect to depends on the number of existing components in the current configuration. In object-oriented systems such as the one described in [33], type hierarchies and inheritance are additional means to support compact, nonredundant knowledge encoding.

Overall, constraint-based approaches are among the most popular ones when it comes to model and solve configuration problems. In particular, because highly efficient constraint solvers are available and because a wide range of configuration and synthesis tasks can be modeled in an intuitive way.

2.3 Distributed Modeling

In scenarios such as the above-mentioned multimarket procurement setting, several business partners co-operate to provide and configure the final customer solution. Due to the above-mentioned reasons of privacy or the need to reuse an existing system, a centralized configurator approach may however not be applicable and a distributed, decentralized solution is required.

From the modeling perspective this means that several configuration models (knowledge bases) are developed. The main problem here is that in many cases the subcomponents of the overall solution, which are provided by different partners or suppliers, cannot be configured independently. A certain parameter setting for the main switching system may in our example restrict the set of possible options of a networking component provided by a supplier. Thus, the participating configurator systems have to exchange information about (parts of) the choices they have taken. Therefore, some parts of the configuration models have to be shared between the systems. Note that the need to integrate different models that cover different aspects of an overall solution can also arise in settings when different "views" (such as sales view and the technical view on the system) are developed during the construction of the system, see [27].

In the configuration system research field, only few works exist that address problems of distributed modeling and problem solving. In [2], Ardissono et al. describe the CAWICOMS workbench, a framework for the development of distributed configurators. The joint provision of VPN services by multiple telecommunication companies serves as an application scenario. Their system architecture assumes a central "main configurator" that communicates with the supplier systems and coordinates the search process. The exchange of information is based on sharing parts of the component-port configuration models based on a shared product ontology. Technically, this means that the configurators share some of their component types. The models themselves are designed graphically using the Unified Modeling Language as knowledge representation mechanism [13] and are automatically compiled into a constraint-based representation of ILOG's JConfigurator [31].

In [17], a formal characterization of the distributed configuration task is given, which is following a consistency-based definition of the central configuration problem [16]. In addition, an outline of an architectural setting for distributed problem solving and a first (theoretical) interaction scheme is given, which is based on the exchange of partial configurations and "conflicts" between the involved agents. From the modeling perspective, again the main assumption is that the involved systems share parts of their configuration knowledge. In the consistency-based approach, they do that by using a common set of predicate symbols and shared domain definitions (component types, ports, etc.).

Outside the configuration research field, approaches to distributed knowledge modeling and reasoning can, in particular, be found in the areas of Multiagent Systems (MAS), ontology development and ontology integration. In common MAS scenarios, several software agents cooperate in order to solve a given problem, e.g., in the areas of logistics, trading, or scheduling. Typical characteristics are autonomously acting agents, no central coordination and that each agent has only a limited view on the overall problem. In order to communicate with each other, agents usually rely on shared knowledge and conceptualizations as well as the use of an agent communication language comprising a set of defined "performatives." Standardization approaches such as the Knowledge Interchange Format (KIF) were developed in the early 1990s. The Foundation for Intelligent Physical Agents (FIPAs) later on defined a wide array of specifications in the field of agent-based systems, in particular in the area of agent communication languages. A detailed review of these modeling approaches is however beyond the scope of this paper.

From a modeling perspective, distributed configuration problems, as described in the previous section, share several characteristics with classical multiagent systems: they are de-centralized, support local views on the problem and rely on a shared ontology and defined communication mechanisms. However, to the best of our knowledge, we are not aware of configurator systems built based on FIPA's standards or KIF. Note that from a problem solving perspective, existing systems such as the one described in [2] often follow an approach where there is a centralized controlling agent which coordinates the other systems, which is not standard in systems composed of autonomous agents.

With respect to representation formats for ontologies, new ontology description languages such as the Web Ontology Language (OWL) emerged with the vision of the Semantic Web.[1] The usage of OWL and the Semantic Web Rule Language for building configuration knowledge bases is discussed, e.g., in [55] and [53]; an in-depth analysis of Semantic Web technologies and their limitations with respect to configuration knowledge representation can be found in [15]. The usage of OWL as a knowledge representation mechanism in distributed configuration is proposed in [20], [21], and [19].

From an engineering perspective, decentralized modeling and ontology development approaches typically require extensive communication and coordination between the involved partners and engineers. For instance, Garcia et al. [24] contains a deeper discussion of methodological approaches to managing this complex task and of existing techniques for ontology alignment, mapping or merging.

3 Distributed Problem Solving

While there is a large body of work on distributed problem solving (e.g., on distributed constraint satisfaction, multiagent planning or agent-based simulation), only few papers can be found that specifically aim at developing algorithms for solving distributed configuration problems. In the following section, we will briefly review these existing approaches before we discuss methods for distributed constraint satisfaction, which is the basis for the distributed GCSP (DisGCSP) method put forward later on in this paper.

3.1 Distributed Configuration Problem Solving

In the above-mentioned CAWICOMS framework [2], two different distributed problem solving algorithms for two application scenarios are used. In both cases, the solution architecture is based on having a centralized main configurator that coordinates multiple "CAWICOMS-enabled" supplier configurators that implement a defined XML-based interface and which share parts of their configuration model. Note that technically, the shared conceptual configuration models in CAWICOMS are automatically translated into the generative and component-oriented CSP representation of ILOG's[2] Java-based JConfigurator library.

The first of the two algorithms, which was developed for the multimarket telecommunication switch scenario, is based on forward checking and *synchronous backtracking*. The main assumption in that scenario is that there exists a tree-structured supply chain and that the participating configurators share individual problem variables. Two types of variable sharing are defined: 1) a variable of one configurator may be "relevant" to one of the suppliers, that is, an assignment of a value has to be communicated to the next lower level on the supply chain. 2) A supplier S may "publish" a variable to the next higher level, with the goal that the upper level configurator asks S in the solution process for a value for this variable.

Based on these definitions, a synchronous distributed backtracking algorithm is proposed. Before trying a value assignment in the search process, each local algorithm first checks if the variable "belongs" to one of the supplier systems. In such a case, it contacts the supplier configurator and asks for a value. If no value can be found, local backtracking is initiated. Likewise, after a variable assignment, the system checks if one of the suppliers has to be informed of the value change. If the new value is not accepted by the supplying system, again backtracking has to take place, see [18]. Similar synchronous strategies are also used in [11] and [20]. Synchronous, client-server style algorithms are in general relatively easy to implement and have—at least in the sketched domains—a good correspondence to the hierarchical real-world problem setting. However, they also have limitations due to their sequential nature and unfocused backtracking behavior, which can lead to a high number of messages to be exchanged among the configurators. Recording of so-called "no-goods" or more advanced backtracking techniques such as backjumping are possible approaches to reduce the search and communication costs.

The second algorithm of the CAWICOMS framework also assumes a hierarchical supply chain but introduces two additional concepts for the application scenario of configuring Virtual Private Networks. First, *parallel problem solving* is supported for situations in which suppliers are able to guarantee that they will be able to find a suitable

1. http://www.w3.org/2004/OWL/.

2. http://www.ilog.com.

subconfiguration for any input. Second, since in this domain the configuration of a subnetwork may also require human interaction, *long-lasting configuration sessions* are supported by the J2EE-based software framework. While this approach supports some form of parallelism, the interaction design is again mainly characterized by a client-server style interaction scheme. Furthermore, it is limited to situations where there are either no interconfigurator constraints or all these constraints are contained in the shared part of the knowledge bases of the involved agents. Finally, questions of obtaining optimal solutions cannot be dealt with easily in such scenarios.

With regard to the communication mechanism, the CAWICOMS framework includes a proprietary API that defines a set of methods to create components, request a solution from the supplier configurator, or to forward variable assignments to the supplier. Note that instead of using a FIPA-standardized communication language, the interface is based on standard Web Services and XML technology. Thus, also legacy configurators not based on JConfigurator technology can be integrated as long as appropriate wrapper components are available that support the defined interface.

A general interaction model for agent communication in distributed configuration scenarios is given in [17]. In this design, a dedicated facilitator agent is introduced that avoids peer-to-peer connections between the involved agents. In particular, the facilitator agent is not only responsible to deliver messages about variable assignments to other agents but in particular also to resolve conflicts or inconsistencies that can arise in the distributed search process. The proposed interaction model is inspired by Distributed CSP (DisCSP) algorithms and the concept of nogood recording and distributed backtracking. Since the basic interaction and conflict resolution strategies more or less represent a "blind-search" approach, Felfernig et al. [17] propose a set of measures to improve efficiency such as the exploitation of the supply chain structure which leads to a reduction of parallelism and therefore also of probability of conflicts.

3.2 Distributed Constraint Satisfaction Problems

In contrast to classical CSPs, in a Distributed CSP, the main assumption is that several agents participate in a joint problem solving process and that the variables and constraints are distributed among agents. The first approaches to formalizing and solving such problems in a distributed and concurrent manner have been proposed in the early 1990s by Yokoo et al. [51].[3] Viewed from a more abstract perspective, the DisCSP formalism as presented by Yokoo et al. can be seen as an infrastructure to model and solve a variety of problems from the area of Multiagent Systems and distributed AI including distributed resource allocation, scheduling or truth maintenance [53]. Correspondingly, we choose DisCSPs as a basis for our approach to modeling and solving distributed configuration problems, in which several agents jointly and in a loosely coupled, asynchronous manner cooperate in the problem solving process.

3. See also [8] for an early discussion of DisCSP problems.

In [52], Yokoo et al. give a precise characterization of the DisCSP approach and propose different distributed algorithms. Beside the above-mentioned assumption that the constraints and variables of the variables are distributed among agents, Yokoo et al.'s framework states the following requirements regarding the communication among agents: 1) agents communicate by sending messages to other agents they know and 2) the message delivery time is unknown but finite and messages are delivered loss-free in the order they have been sent.

In the following, we summarize the characteristics of asynchronous search algorithms such as Asynchronous Backtracking (ABT) [51], adapted in a way that they allow agents to know only the constraints that they enforce, see also [57].

1. $A = \{S_1, \ldots, S_n\}$ is a set of n totally ordered agents (representing the participants in the distributed configuration problem in our case). S_i has priority over S_j if $i < j$.
2. Each agent S_i owns one single variable[4] and knows all the constraints which involve its own variable and the variables of higher priority agents.[5] The constraints known by S_i are referred to as its local constraints, denoted Γ^{S_i}. S_i is said to be *interested in* those variables which are contained in its local constraints. A *link* exists between two agents if they share a variable. Links are directed from the agent with higher priority to the agent with lower priority. Links from agent S_1 to agent S_2 are referred to as an *outgoing links* of S_1 and *incoming links* of S_2.
3. A variable assignment is a pair (x_j, v_j), where x_j is a variable, and v_j a value for x_j from its domain.
4. The *view* of an agent S_i is a set of the most recent assignments, which agent S_i received for those variables it is interested in.
5. The agents communicate using a defined set of messages.

The following message types are used.

ok?: Agents with higher priorities communicate value assignments for variables via **ok?** messages to the lower priority agents.

nogood: In situations where an agent cannot find an assignment that does not violate its own constraints and its already known nogoods, it generates an explicit nogood $\neg N$. A nogood can be seen as a constraint that forbids a certain combination of value assignments to a set of variables. The **nogood** message is sent to the agent with the lowest priority which has proposed an assignment for one of the variables in N.

addlink: With this message, the receiving agent is informed that the sender is interested in its variable. A *link* is always established from a higher priority agent to a lower priority agent.

Based on these definitions, Yokoo et al. propose two noncentralized problem solving algorithms [52]: Asynchronous

4. We will see later, how one can see this variable as a tuple of variables which are treated simultaneously.

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

Backtracking and Asynchronous Weak-Commitment Search (WCS).

ABT is the basis for our configuration problem solving approach and roughly works as follows: for ABT, the DisCSP is seen as a graph where agents (and their single variable) are the nodes and interagent constraints are directed links between these nodes. One of the agents involved in each (binary) constraint is designated as the "owner" and evaluator of the constraint. The other agent involved in the constraint informs the owner whenever it assigns a new value to its variable, so that the owner can check the consistency of the new value with the constraint. The set of values each agent received over the incoming links is called the *agent_view*. For the realization of the distributed backtracking process, each agent has a priority value, which can be simply based on the alphabetical order of the variable names.

When the algorithm starts, each agent assigns a value to its variable independently of the others and sends an **ok?** message to the neighbors. Upon receipt of an **ok?** message, an agent revises its own assignment if it is different from the one of a higher priority agent. If no such assignment can be found, a new constraint (nogood) is generated and returned via a **nogood** message to the higher priority agent, who in turn will try a new assignment. Infinite processing loops are avoided by the priority ordering. More details about the algorithm and proofs of soundness and completeness are given in [52].

Weak-Commitment Search is an alternative backtracking algorithm, which achieves an improved runtime efficiency compared to ABT by dynamically determining agent priorities and relying on a min-conflict heuristic to select the next variable value. When multiple values are consistent with the agent view, it chooses the one that minimizes the number of constraint violations with lower priority agents. In addition, WCS follows a strategy to start with a partial solution for some variables, which is incrementally extended until a solution is found. If no solution is found, the partial solution is abandoned and the search restarts with another partial solution. In order to ensure algorithmic completeness WCS buys its runtime efficiency improvement at the expense of an exponential space requirement in the worst case. However, in supply chains a natural prioritization among agents is given due to the hierarchical orderer/contractor relationships. As WCS with static agent ordering basically resembles an ABT algorithm, we rely on ABT in our work.

In recent years, further problem solving schemes or extensions to handle multiple variables per agent have been proposed. Among them are a distributed break-out approach [54], handling of partial solutions for overconstrained problems and the application of techniques, which have been successfully applied for centralized CSPs (such as maintaining arc consistency or value aggregation) [40]. Furthermore, Bessière et al. [5] show how ABT can be adapted to networks where not all agents can directly communicate to one another. Havens [28] 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 [56], [47], and [48]. In [45], Silaghi et al. achieve 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.

4 Distributed Generative Constraint Satisfaction

In this section, we present technical details of our framework for modeling and solving distributed configuration problems based on the CSP paradigm, see also [57]. We combine the concepts of Generative CSPs (GCSP) and Distributed CSPs in order to support both intuitive and compact configuration knowledge modeling and supply chain integrated, cooperative solution search.

Our approach has the following main characteristics. First, the constraints (including nogoods) are generalized to *generic constraints*, that is, they apply to all instances of variables of a certain type and not to the identities of variables. Second, 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. Note that in previous DisCSP frameworks, the number of variables existing within the system are predetermined. Third, as we are interested in modeling configuration problems according to the component-port approach, some variables model possible connections and depend on the existence of components that could later be connected. As a side aspect, also the domains of the variables may vary dynamically. Note, that the use of generic constraints also allows us to treat the aforementioned second and third aspect more elegantly. Finally, we show how these extensions impact the design of asynchronous distributed problem solving algorithms.

In the following sections, we will first motivate our work with a more detailed example. Then, we will formally define the local Generative CSP and the distributed Generative CSP (DisGCSP) including the required extensions for distributed problem solving.

4.1 Motivating Example

Consider the following typical configuration problem from [12]. The goal is to configure a telecommunication switching system which can support plug-in modules and where problem specific constraints to describe the legal combinations of module types, their capacity as well as individual parameters of the components exist.

Fig. 1 shows a problem in which a system can have modules of different types (A-, B-, and C-modules) and provide possible connection points (ports) for the modules. In our example, we only show a relatively small subset of a larger configuration problem of a technical system, which is indicated by the dotted lines. The problem-specific constraints state that subsystem 1 (placed in the upper part of the figure) can consist of A- and B-modules whereas subsystem 2 may have only A- and C-modules plugged in. The A-modules are "shared" and act as an interface between the two systems.

Each module has an additional parameter describing if it is active or inactive. The left-hand side of Fig. 1 shows an initial situation, where the customer requirement is that

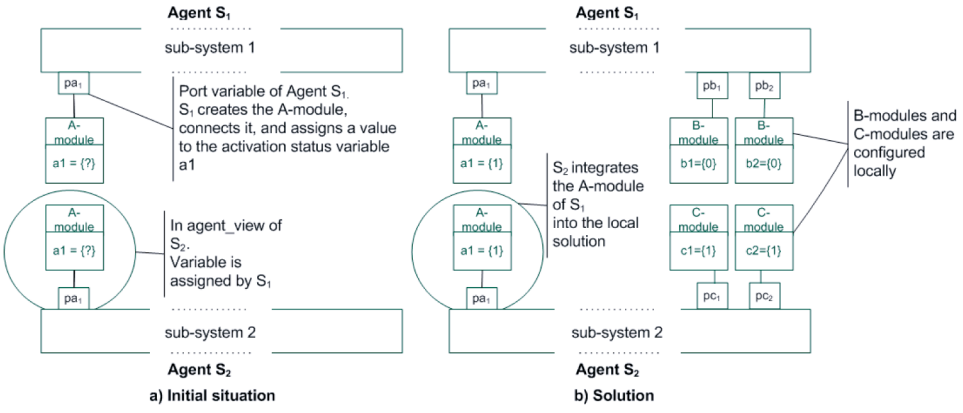

Fig. 1. Example problem.

TABLE 1
Example Constraints for Configuration Problem

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

the configuration has to contain at least one A-module that is connected via a port to the system. According to the constraints described in the following section, the corresponding solution shown on the right hand side includes two additional modules of type B and C. In addition, all A- and C- modules in the configuration are set to active and all B-modules are set to inactive.

In our scenario the overall solution not only consists of sub-systems but due to some technical or organizational reasons they have to be configured by different agents. We formalize this configuration problem as a CSP as follows: each port and each module is represented by a variable.[6] The exact number of problem variables is, however, not known from the beginning. Therefore, constraints cannot be directly formulated on concrete variables. Comparable to programming languages, we thus introduce the concept of variable types. These types allow us to associate a domain to each newly created variable. Furthermore, we specify the relationships between variables in terms of *generic constraints*. In [41], a generic constraint γ is defined as a constraint schema, where metavariables M_i act as placeholders for concrete variables of a specific type t, which is denoted by the predicate $type(M_i) = t$. Note that the subscript i allows us to distinguish between different metavariables in one constraint. The exact semantics of generic constraints will be given in Definition 2 in Section 4.2.

In our example, there are seven different types of problem variables, which represent the ports for the three different module types (t_{pa}, t_{pb}, t_{pc}), the activation status for each module type (t_a, t_b, t_c)[7] as well as a type (t_{ct}) of variables (x_{type}) which count the number of instantiations of each type. Beside the variables, also the specific configuration problem constraints are distributed among the agents and each agent possesses a set of local constraints Γ^{S_i}.

Let us assume that there are two agents who know the following constraint sets, which are defined in detail in Table 1: $\Gamma^{S_1} = \{\gamma_1, \gamma_2, \gamma_5, \gamma_7, \gamma_9\}$ and $\Gamma^{S_2} = \{\gamma_3, \gamma_4, \gamma_6, \gamma_8, \gamma_{10}\}$.

In the example, we omit the general constraints which ensure that whenever a port variable is assigned a value, the corresponding connected component variable must also exist. Based on such generative constraints, the search space can be continuously extended by the instantiation of additional problem variables during the search process. The ultimate goal is (as usual) to find an assignment to all variables such that all the constraints of each agent are satisfied.

6. Note that the *subsystem* components themselves are not explicitly modeled, but only via their characterizing port variables.

7. These variables are shown in Fig. 1 as $a1, b1$, etc.

As mentioned above, the *agent_view* contains the problem variables which are shared between the agents. In our example, γ_6 and γ_{10} are such interagent constraints. They require agent S_2 to have knowledge about all A-modules and their associated port variables. Overall, a solution to a generative constraint satisfaction problem thus requires not only finding valid assignments to the variables, but also determining the exact size of the problem itself. In the next sections, we will develop these concepts of generative CSPs more formally and then detail the required extensions to an existing DisCSP algorithm.

4.2 A Formal Characterization of Generative CSPs

Based mainly on the specific requirements of the configuration domain, the Generative CSP framework was proposed in [26], [41], in particular because the required dynamic creation and addition of components exceeds the capabilities of previous Dynamic or Conditional CSP approaches of [36], [39] or [9]. In the following, we provide a formal definition of a GSCP, which abstracts from the configuration task specific formulation and applies to the wider range of synthesis problems, see also [57].

Definition 1 (Generative CSP). *We define a generative constraint satisfaction problem as a tuple GCSP(X, Γ, T, Δ), where*

- *X is the set of problem variables of the GCSP and $X_0 \subseteq X$ 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 $t_i \in T$ 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.*
- *Δ 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 ith variable of type t. The function $type(x)$ accesses Δ and returns the type $t \in T$ for x and the function $index(x)$ returns the index of x.*

In configuration problems that follow the component-port model [35], the variables which represent the ports (named connections) require a special treatment. The values that are assigned to these variables correspond to references to other components (their IDs, respectively). Since we allow the number of components in a configuration to be determined dynamically, e.g., in order to make a previously unsolvable CSP solvable, we also need the concept of *extensible domains*. The current domain of a port variable is therefore no longer fixed but determined dynamically depending on the status of the configuration process.

Consider the following example of the usage of dynamic domains and counter variables.

Example. Let t_a be the type of the variables which represent instances of *A-modules* and t_{pa} be the type of the (port) variables which can be connected to them. The domain of the *pa*-variables $dom(t_{pa})$ therefore includes references to all variables that represent *A-modules*. We can specify that fact by defining $dom(t_{pa}) = \{1, \ldots, ub\}$, where ub is an upper bound 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., $type(M_1) = t_{pa} \wedge val(M_1) \leq x_{t_a}$.

For our introductory example shown in Fig. 1, we can formalize the local GCSP of agent S_1 in the initial situation (see Fig. 1) as follows:

$$
\begin{aligned}
X^{S_1} &= \{x_a, x_{pa}, x_b, x_{pb}, x_{ct}, a_1, pa_1\},\\
\Gamma^{S_1} &= \{\gamma_1, \gamma_2, \gamma_5, \gamma_7, \gamma_9\},\\
T^{S_1} &= \{t_{ct}, t_a, t_{pa}, t_b, t_{pb}\} \text{ and}\\
\Delta^{S_1} &= \{(x_a, (t_{ct}, 1)), (x_{pa}, (t_{ct}, 2)), (x_b, (t_{ct}, 3))(x_{pb}, (t_{ct}, 4)),\\
&\quad (x_{ct}, (t_{ct}, 5)), (a_1, (t_a, 1))(pa_1, (t_{pa}, 1))\}.
\end{aligned}
$$

With the help of the $index$ function we can reference concrete individual variables. The function $index^{S_1}(t_a)$ therefore returns 1, which indicates that a_1 is the first A-module instance. The domains of variables are consequently defined as

$$
\begin{aligned}
dom(t_a) = dom(t_{pa}) &= dom(t_b) = dom(t_{pb})\\
&= dom(t_{ct}) = \{1, \ldots, ub\},
\end{aligned}
$$

where the domains for the port variables can be additionally limited by domain-specific constraints (e.g., γ_1).

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

In general, constraints are applied to all instances of a specific type, but they can also formulate restrictions on specific initial variables from X_0 using the $index$ function.

Next, we define the concept of consistency of generic constraints. Let us consider a GCSP(X, Γ, T, Δ) and a generic constraint $\gamma \in \Gamma$ which restricts the metavariables $M_a, \ldots, M_k$. Furthermore, let $type(M_i) \in T$ be the defined variable type of the metavariable M_i. The consistency of generic constraints is defined as follows:

Definition 3 (Consistency of Generic Constraints). *Given an assignment tuple θ for the variables X, then γ is said to be satisfied under θ, iff $\forall x_a, \ldots, x_k \in X : type(x_a) = type(M_a) \wedge \ldots \wedge type(x_k) = type(M_k) \rightarrow \gamma[M_a|_{x_a}, \ldots, M_k|_{x_k}]$ is satisfied under θ, where $M_i|_{x_i}$ indicates that the metavariable M_i is substituted by the concrete variable x_i.*

A *generic* constraint must be considered as a constraint scheme which is expanded into a set of constraints after a preprocessing step. In this step, metavariables are replaced by all possible combinations of concrete variables of the same type. Let us, for example, assume that a fragment of the GCSP of agent S_1 (excluding counter and port variables) is defined as follows:

$$X^{S_1} = \{a_1, b_1, b_2\}, T^{S_1} = \{t_a, t_b\} \text{ and}$$
$$\Delta^{S_1} = \{(a_1, (t_a, 1)), (b_1, (t_b, 1)), (b_2, (t_b, 2))\}.$$

In such a setting, the satisfiability of the generic constraint γ_9 can be 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 GCSP). *Given a GCSP(X_0, Γ, T, Δ_0), then its solution consists of a set of variables X, type and index assignments Δ and an assignment tuple θ for the variables in X, such that*

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

In our definition for a solution to a given GCSP, intentionally no minimality criterium regarding the number of variables is included. Note, that in practical applications in many cases domain-specific optimization criteria exist such as the total cost or the future flexibility of the solution. Thus, nonminimal solutions can be preferred over minimal ones in practice.

Going back to our example from the telecommunication switches domain, a possible solution (excluding counter variables) for the local GCSP of agent a_1 could be as follows:

$$X^{S_1} = \{a_1, pa_1, b_1, b_2, pb_1, pb_2\},$$
$$\Delta^{S_1} = \{(a_1, (t_a, 1)), (pa_1, (t_{pa}, 1)), (b_1, (t_b, 1)),$$
$$(b_2, (t_b, 2)), (pb_1, (t_{pb}, 1)), (pb_2, (t_{pb}, 2))\}$$

and the assignment tuple ($a_1 = 1$, $pa_1 = 1$, $b_1 = 0$, $b_2 = 0$, $pb_1 = 1$, $pb_2 = 2$), where $b_1, \ldots, b_2$ and $pb_1, \ldots, pb_2$ are the names of *generated* variables.

With respect to practical implementations of the GCSP approach, note that the names for generated variables are unique and can be randomly chosen by the GCSP solver implementation. Constraints therefore must not formulate restrictions on the variable names of the generated variables. Consequently, the substitution of any generated variable (i.e., $x \in X \setminus X_0$) by a newly generated variable of the same type, index and value assignment has no effect on the consistency of the generic constraints.

Compared to previous works, our GCSP definition generalizes the definition from [41], in which a configuration-specific problem definition was chosen and only variables of a single type, i.e., component variables, can be created. In contrast, we assume that a finite set of variable types T is given and during the problem solving process variables having any of these types can be generated. A similar approach can be found in current CSP implementations of configuration systems (e.g., [33] or [12]), which use a type system for problem variables, where new variable instances are dynamically created. This is only indirectly reflected in the definition of [41] by the domain definition of component variables, which we explicitly represent here as a set of types.

Our definition of the GCSP is also more general than previous approaches with respect to knowledge representation. In our definition of *generic constraints* we do not require the use of a specific language for the formulation of restrictions. Examples of previous systems which use specific constraint languages are the LCON language used in the COCOS project [41] or the configuration language (API) of the ILOG Configurator [33].

A particular aspect of GCSP formulations is that the set of variables X can be theoretically infinite which in turn can lead to an infinite search space. Real-world GCSP solver implementations therefore usually put a limit on the total number of problem variables to ensure decidability and finiteness of the search space. Given such an upper limit, a GCSP could be theoretically modeled as a dynamic CSP and finally even as a CSP which includes complex constraints to ensure activation and de-activation of variables during the search process. However, formulating large and complex configuration problems this way would be mostly impractical from the perspective of knowledge representation and maintenance, which is crucial for the success of knowledge-based applications such as configuration systems.

5 Extending the Distributed CSP

In this section, we will formally describe how to extend an existing DisCSP approach described in Section 3.2 for the class of generative CSPs proposed in the previous section; see also [57]. In our work, we focus on the ABT algorithm, which in contrast to heuristic approaches is also capable of finding optimal solutions and for which various performance extensions exist.

We will first summarize the properties of the ABT algorithm that guarantee its correctness and completeness as described in [51]. Then we apply the general DisCSP framework to scenarios in which each agent has to solve a local GCSP. In principle, our formalization and correctness guarantees for DisGCSPs are based on the idea of viewing the DisGCSP-case as a series of DisCSP problems. Each time an agent extends the solution space of his local GCSP by creating an additional variable, the problem setting is transformed into a new DisCSP setting, which again has all the properties required by ABT for correct functioning.

We view a distributed configuration problem as a multiagent scenario, where each agent wants to satisfy a local GCSP. Agents keep their constraints private for security and privacy reasons but also share variables to others. Because constraints in our settings also employ metavariables, the *interest* of an agent in variables has 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 metavariables $M_a, \ldots, M_k$, where $type(M_i) \in T^{S_j}$ is the variable type of the metavariable M_i, and $\exists M_i \in M_a, \ldots, M_k : type(x) = type(M_i)$.*

Definition 6 (Distributed Generative CSP). *We define a distributed generative constraint satisfaction (DisGCSP) problem as follows:*

- *$A = \{S_1, \ldots, S_n\}$ 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.*

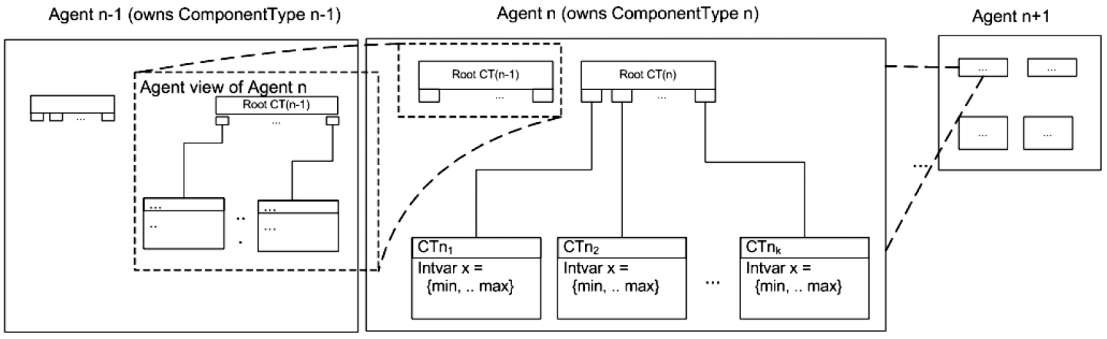

Fig. 2. Benchmark configuration problem.

- *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, it must hold that $x \in X^{S_i}$.*
- *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 which indicates that they share variable x. As usual, the *link* is directed from the agent with higher priority to the agent with lower priority.

Definition 7 (Solution to a DisGCSP). *Given a DisGCSP involving a set of n agents, its solution consists 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 θ^{S_i} are a solution for the local $GCSP^{S_i}$ of agent S_i.*

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

For communicating variable assignment and nogood messages between agents in the distributed scenario, we introduce the concepts *generic assignment* and *generic nogood*.

Definition 8 (Generic Assignment). *A generic assignment is a unary generic constraint. It has the form: $\langle M, i, v \rangle$, where M is a metavariable, i is a set of index values for which the constraint applies, and v is a value.*

Definition 9 (Generic Nogood). *A generic nogood has the form $\neg N$, where N is a set of generic assignments for distinct metavariables.*

Similar to the original DisCSP formulation, value assignments to variables are communicated among agents via **ok?** messages. In the GCSP case, messages however transport *generic assignments*, which represent domain restrictions on variables by unary constraints. Each of these unary constraints in our DisGCSP has an unique identifier attached which is called constraint reference (cr) [46]. Any inference has to attach the crs associated to arguments into obtained nogoods. We treat the extension of the domains of the variables as constraint relaxation, see [46]. For this purpose, we extend the existing ABT algorithm with the following features.

- An **announce** message broadcasts a tuple (x, t, i) to all agents, where x is a newly created variable of type t and with index i. The receiving agents determine their interest in variable x and react depending on their interest and priority in one of the following ways: 1) send an **addlink** message transporting the variable set $\{x\}$, 2) add the sending agent to the outgoing links, or 3) discard the message.
- A **domain** message broadcasts a set CR of obsolete constraint references. Any receiving agent removes all the nogoods which have a constraint reference $cr \in CR$ attached to them. The message receiver additionally calls the function *check_agent_view()* (see [51]), to make sure that the assignments in the agent view are consistent or otherwise generate a nogood.
- A **nogood** message transports *generic nogoods* $\neg N$ which contain assignments for metavariable instances. These messages are multicasted to all agents interested in $\neg N$. An agent S_i is interested in a generic nogood $\neg N$ if it has *interest in* any metavariable in $\neg N$.
- When an agent has to revoke the creation of a new variable due to backtracking in the local solving process, it assigns a specific value from its domain to the variable which indicates its deactivation and communicates this fact via an **ok?** message to all interested agents.

Note that in practical implementations as described later on, a broker agent can be introduced in order to avoid too many messages. Such a broker can maintain a static list of the agents as well as their interest in variables of specific types. Using such an implementation, the agent who created a new variable only has to request a list of interested agents from the broker agent and does not need to broadcast an **announce** message to all agents.

A discussion and proof of the correctness and completeness of the proposed extensions for the ABT scheme are given in [57].

6 Evaluation

In order to evaluate the applicability of the proposed GCSP-based approach to configuration, we developed a Java-based framework that implements a distributed and asynchronous GCSP-algorithm. For solving the agents' local problems, ILOG's *JConfigurator* library was used as a basis.

6.1 Benchmark Problem

For the determination of execution times for generative and distributed CSP solving, we used the parameterizable configuration problem sketched in Fig. 2. In contrast to previous works in distributed constraint satisfaction problems we do not assume that each configuration agent "owns" only one single variable but is responsible for the configuration of one component type of which multiple instances can exist. In Fig. 2, Agent n is therefore responsible for component type CTn of which up to k instances can be created. The *local configuration problem* for Agent n consists of creating a sufficient number of component type instances of type CTn and connecting them to a so-called root component, which is part of every configuration problem when using the *JConfigurator* library. For each component type instance, the value of a single integer variable has to be determined as part of the configuration problem.

In order to create a distributed problem setting, a given number of agents of the same basic structure is created. In addition to its local problem, each agent n can also have a view or *interests* on the component type instances of neighboring agents. For instance, Fig. 2 depicts the interest of Agent n in a component of its left neighbor. To test different problem classes, we both evaluated scenarios where the agent only had a view on the left neighbor's variables as well as scenarios, in which also the right neighbor's variables were visible. Note that the inclusion of a view on the left neighbor's variables as shown in Fig. 2 means that whenever Agent $n-1$ determines a local solution and creates instances of its component type and assigns variable values, it informs Agent n of the new solution. Adding also variables from additional neighbors to one's agent view ultimately means that higher numbers of messages need to be passed for problem solving.

In order to evaluate the performance of our algorithm, we designed three different problem scenarios with a varying solution complexity.

- In Constraint Scenario CS1, one generic constraint is added for each component type that states that the variables of the local component type must be *different* from the value of the left neighbor's component type. In this scenario, no distributed backtracking is required because every agent is able to find a solution that is compatible with his left neighbor's solution.
- In Constraint Scenario CS2, one generic constraint for every second component type is added which states that the variable value of the component type must be different from the left *and* the right neighbor's value.
- Constraint Scenario CS3 is designed to be a very hard and ill-structured case. It is similar to Scenario CS1, however the only generic constraint states that the integer variable of the left neighbor has to be greater than the variable value of the right neighbor. The distributed problem solving is particularly inefficient, when using the default variable value selection strategy that starts with the lowest value of the domain. This means that all agents start in parallel and try to assign the smallest value (zero) first. Assume a scenario with $n = 10$ agents. The smallest possible value for Agent 1 will be 9. Agent 1 will however incrementally try values from 0 to 9 first and communicate them to the right neighbor (who in turn will only incrementally increase the variable value stepwise). Overall, the distributed system will continuously backtrack until a solution is found where Agent-1 assigns 9 to the variables, Agent 2 has 8 as a value and so on.

Overall, our problem setting can be varied based on the following parameters: number of agents, minimum number of instances per component type, size of the domain of the component variables (at least equal to the number of agents for constraint scenario CS3), and inclusion of the right neighbor's variables into the agent view.

6.2 Algorithm and Implementation Details

Algorithm. As described in previous sections, our system implements a variant of Yokoo et al.'s Asynchronous Backtracking algorithm [52], in which each of the cooperating agents maintains a local *agent view* and communicates local variable assignments through **ok?** messages and inconsistencies of variable assignments through **nogood** messages. The communication between the agents in our system is based on a central mediator component that dispatches the messages between the agents. Each agent maintains a queue of incoming messages which are processed sequentially by their order of arrival. The distributed and asynchronous nature of the setting is simulated in our framework by running each agent in a separate execution thread.

In general, the algorithm is designed in a way that all agents start their local problem solving process in parallel. In an infinite loop, the agents first solve their local problem, inform the other agents that have interest in individual variables and then suspend execution until **ok?** or **nogood-**messages arrive.

The distributed algorithm stops when 1) no solution can be found, which happens if one of the agents cannot determine another nogood in the backtracking phase, or 2) no more messages are exchanged and all agents are "stable," which is monitored in our framework by the mediating agent.

Note, that in our framework we used a basic variant of Yokoo et al.'s sound and complete ABT algorithm and did not rely on enhancements such as dynamic agent ordering [1] or Aggregation Search [45]. Integrating such techniques as well as additional heuristics that are specific for configuration problems remain part of our future work.

Nogood generation. In constraint satisfaction problems, a conflict or nogood corresponds to a partial or complete value assignment constellation which is not compatible with the constraints of the problem. In the distributed problem setting, each agent maintains a list of known nogoods and thus removes these constellations from the local solution space.

In our distributed configuration problem, a nogood corresponds to a partial or complete configuration that violates one or more of the generic constraints of an agent. In our framework, we rely on the explanation facility of the *JConfigurator* library, which implements Junker's QUICKXPLAIN algorithm [32]. This algorithm is capable of calculating

Encoding	Nbr. of agents	Possible nbr. of inst.	Min. nbr. instances	Overall time	Explan. time	Check-time	NG/Back-tracks	Msgs	Checks
DisGCSP	3	2	2	< 0.1	0	< 0.1	0	3	7
DisCSP				< 0.1	0	< 0.1	0		
DisGCSP	5	3	3	< 0.1	0	< 0.1	0	10	24
DisCSP				<0.1	0	<0.1	0		
DisGCSP	8	3	3	< 0.1	0	< 0.1	0	27	60
DisCSP				<0.1	0	<0.1	0		
DisGCSP	10	5	3	<0.1	0	<0.1	0	45	97
DisCSP				0.11	0	<0.1	0		
DisGCSP	15	10	10	0.16	0	<0.1	0	104	223
DisCSP				0.5	0	0.11	0		
DisGCSP	15	15	10	0.22	0	<0.1	0	102	219
DisCSP				0.86	0	0.21	0		
DisGCSP	20	5	3	0.18	0	<0.1	0	189	398
DisCSP				0.39	0	<0.1	0		
DisGCSP	25	10	8	0.43	0	<0.1	0	303	612
DisCSP				1.3	0	0.22	0		

Fig. 3. Results for scenario CS1.

minimal conflicts caused by constraint propagation in a highly efficient way by recursively partitioning the problem into subproblems. In our configuration problem setting, a conflict can be caused by an inconsistent assignment to the variables of the components or by the fact that an inconsistent number of components exist in the configuration.

In our framework, the components of the partial configuration (including the variable assignments) that are part of the minimal conflict are sent to the appropriate agent in a **nogood** message. Note that the receiving agent generates a *generic* nogood from this partial configuration, because the component instances in our problem setting are interchangeable.

6.3 Evaluation and Discussion

The goals of our experimental evaluation are not only to demonstrate the general applicability of the distributed GCSP approach for configuration problems but also to analyze the benefits of the generative approach and the exchange of generic nogoods in comparison to a distributed, nongeneric CSP algorithm.

6.3.1 Variation of Problem Sizes

In order to test the general feasibility of our approach, we analyzed the algorithm's runtime behavior for different problem sizes. We, thus, varied the number of involved agents and the minimum number of component instances per component type which corresponds to increasing the number of problem variables. Among other figures, we measured for each scenario the total execution times, the time needed for generating explanations and the number of distributed backtracking steps.

In particular, we also consider the number of required message exchanges to be relevant in practical scenarios because communication with remote configurators can be a costly operation. In a variation of the benchmark problem setting, we included also the right neighbors of each agent in the agent view, which does not actually influence the solution space but rather requires more messages to be exchanged.

Note that the local configuration problems of our benchmark scenario are rather underconstrained, which means that local solutions can be very quickly calculated by the *JConfigurator* library. In our experiments we intentionally did not vary or increase the complexity of the local configuration problem because we want to focus on the costs for making the assignments of the shared variables consistent and the number of required message exchanges. Increasing the complexity of solving the local configuration problem would of course increase the execution time, but this increase would be application-specific.

6.3.2 Comparison with Traditional CSP-Encoding

As described in previous sections, dynamic or generative CSPs have an advantage over classical CSPs, at least with respect to the compactness of the knowledge base. To compare the generic, distributed approach with a traditional distributed CSP, we also automatically generated a nongeneric encoding for the different problem variations described above. Thus, instead of allowing multiple instances of each component type at an agent, we created for each possible instance an artificial component type, of which exactly one instance is generated. In addition, instead of using generic constraints, for each component instance, individual constraints on the shared variables are added to the CSP. Correspondingly, the exchanged nogoods are no longer generic and relate to one specific component instance. Note that in comparison with the GCSP-approach, exactly the same algorithm is used but only the problem encoding has been changed.

6.3.3 Notes on Measurements

In the subsequent figures, the results of the different experimental configurations are shown. The execution times, number of message exchanges and backtracking operations are averaged over 10 runs of the algorithm, because the runtime behavior of the distributed algorithm is not deterministic. The experiments were run on a Windows 7-based standard laptop computer with 4 GB of RAM and an Intel Core I5 processor.

Results for scenario CS1. Fig. 3 shows the detailed statistics for various constellations of the first, backtrack-free scenario. The columns of the table hold the following values.

- Nbr. of agents: Number of agents in the specific problem instance.
- Possible nbr. of inst: Maximum number of component type instances per agent.
- Min. nbr. instances: Minimum number of instances per agent to be instantiated.

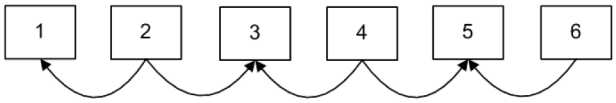

Fig. 4. Overview of scenario CS2.

- Total time: Average execution time (in seconds) to solve the problem.
- Explan. time: Average time needed per agent (in seconds) for nogood creation/explanations.
- Check-time: Average time needed per agent (in seconds) to check the consistency of assignments and to find local solutions.
- Msgs: Number of messages exchanged between agents.
- Checks: Total number of consistency checks or solution searches of all agents.

Remember that scenario CS1 represents a distributed configuration scenario, in which the participants (of the supply chain) propagate their parameter choices or partial configurations to their partners who take these settings into account when configuring their own subassembly. The assumption of scenario CS1 is that every agent will always be able to find such a compatible solution, that is, distributed backtracking is not necessary. Note, that such an assumption was also made for the second phase of the Virtual Private Networks configuration scenario described in [2].

Not surprisingly, the overall execution times to solve such distributed configuration problems are very low, that is, below 1 second for all but one problem instance. Since there is no backtracking involved, the time required for generating nogoods (explanation time) is zero. For the same reason, the number of required messages and solution searches (checks) is identical both for the generative approach and the traditional DisCSP approach. What can be observed, however, is that the total execution times are slightly higher for the nongeneric approach as more individual constraints (instead of generic ones) have to be considered in the search process. Overall, the main advantage of the GCSP approach in such a backtrack-free setting can be found in the more compact way the local configuration problems are modeled and the slightly more efficient local problem solving.

Results for scenario CS2. The general structure of this supply chain scenario with multiple hierarchies is shown in Fig. 4. In the example, Agent 2 forwards its local configuration to its neighbors Agents 1 and 3. In addition, the constraints of the middle or parent agent (e.g., Agent 2) state that its variable values must be different from that of Agent 1 and that in addition the variable values of Agents 1 and 3 must also be different. Given these interagent dependencies and the incremental variable value selection strategy, distributed backtracking is required.

The results of several different problem instances of scenario CS2 are shown in Fig. 5. Again, we vary the number of involved agents and the number of possible and required component type instances. Note that a scenario in which more than five companies or configuration systems interact to build a customer solution represents a rather complex constellation in practice.[8]

8. The local configuration problems are of course more complex in reality, which is, however, not the target of this evaluation.

The results show that with the generative DisGCSP approach all defined problem settings could be solved by the agents in less than one second. Given the design of the scenario, no *distributed* backtracking is required for the smallest problem instance with only three agents.

Scenario CS2 additionally reveals that the DisGCSP approach is preferable over the DisCSP approach with the traditional CSP encoding. Thanks to the usage of generic constraints and nogoods and the interchangeability of the component type instances in the GCSP model, the number of required messages, distributed backtracking phases and searches for local solutions is significantly lower for our DisGCSP approach.

Results for scenario CS3. Fig. 6 finally shows the numbers for the hard scenario CS3 in which the variable values of the component instances of an agent must be smaller than the values of its left neighbors, which means that at the end, the variable values are ranked in decreasing order. Given n agents, Agent k will assign value $n - k + 1$ to the variables. This fact and the incremental variable value selection strategy lead to significant backtracking. When viewed as a supply chain scenario, this setting would represent a situation in which we have a sequential supply chain of depth n. In practice, supply chains with configuration dependencies of more than three levels might not exist very often. However, we include the scenario in order to evaluate our DisGCSP approach also in such hard problem settings.

In the first four scenarios in Fig. 6, we again incrementally added more agents and component instances. We can observe that the execution times quickly increase also for the DisGCSP approach. For the eight-level supply chain setting, the algorithm nearly needs 3 minutes. Using the classical DisCSP approach, already the smaller six-agents setting required nearly half an hour; the next larger one with eight agents did not complete within a 1-hour time limit. The next scenario in Fig. 6 (with 10 agents) was performed to measure whether the number of agents or the number of shared component type instances has a greater impact on execution time. It shows that with the DisGCSP approach, such problems can be solved comparably fast because the interagent dependencies are reduced. With the classical DisCSP approach, however, no solution could be found within the given time limit.

The last two rows of Fig. 6 finally show the effect of fixing the number of existing components in a configuration. In these scenarios the maximum number of instances is set to be equal to the minimum number of components, which in turn means that the configuration system does not need to reason about component existence. The measurements for these cases show that the solving process can be clearly shortened for the DisGCSP approach when dynamic component creation is not required. Here, we report examples for even more complex supply chain settings, which can be solved in about one second with the help of generic constraints and nogoods. When following the classical DisCSP approach, these problems were not solvable within the given time limit.

Overall, the reported experiments clearly demonstrate the advantage of a generic approach to distributed constraint satisfaction for configuration problems both with respect to total execution time and with respect to the number of required message exchanges, which can be a limiting factor in practical settings.

Encoding	Nbr. of agents	Possible nbr. of inst.	Min. nbr. instances	Overall time	Explan. time	Check-time	NG/Back-tracks	Msgs	Checks
DisGCSP	3	2	2	<0.1	0	<0.1	0	8	14
DisCSP				<0.1	0	<0.1	0	8	14
DisGCSP	4	5	2	<0.1	<0.1	<0.1	3	27	45
DisCSP				0.17	<0.1	<0.1	14	64	104
DisGCSP	8	5	3	0.12	<0.1	<0.1	12	78	122
DisCSP				2.1	0.17	0.56	170	662	1011
DisGCSP	10	7	5	0.20	<0.1	<0.1	16	106	164
DisCSP				7.6	0.55	1.75	365	1480	2210
DisGCSP	12	10	8	0.35	<0.1	0.1	17	139	210
DisCSP				35	2.3	6.8	966	3760	5631
DisGCSP	15	10	8	0.43	<0.1	<0.1	21	172	260
DisCSP				96	3.3	11.55	1838	6892	10432
DisGCSP	20	5	3	0.38	<0.1	<0.1	38	281	421
DisCSP				7.2	0.41	1.2	595	2444	3641
DisGCSP	25	10	8	0.92	<0.1	0.12	51	363	543
DisCSP				126	6.0	18.9	3228	12957	19153

Fig. 5. Results for scenario CS2.

Encoding	Nbr. of agents	Possible nbr. of inst.	Min. nbr. instances	Overall time	Explan. time	Check-time	NG/Back-tracks	Msgs	Checks
DisGCSP	3	2	2	<0.1	<0.1	<0.1	3	8	20
DisCSP				<0.1	<0.1	<0.1	10	23	51
DisGCSP	4	5	2	<0.1	<0.1	<0.1	6	15	36
DisCSP				7.8	0.8	1.8	525	1086	2136
DisGCSP	6	5	3	8.6	1.53	0.58	58	124	300
DisCSP				1794	99.4	402	14566	30076	58349
DisGCSP	8	5	3	165	38	10.3	242	506	1225
DisCSP				*	*	*	*	*	*
DisGCSP	10	3	2	<0.1	<0.1	<0.1	8	21	42
DisCSP				*	*	*	*	*	*
DisGCSP	15	5	5	0.35	<0.1	<0.1	62	140	334
DisCSP				*	*	*	*	*	*
DisGCSP	20	7	7	1.4	0.12	0.13	186	403	970
DisCSP				*	*	*	*	*	*

Fig. 6. Results for scenario CS3.

7 Summary

In this work, we proposed a constraint-based approach to modeling and solving distributed configuration problems. After reviewing previous approaches to distributed configuration problem solving and reasoning, we provided a formal definition for Generative Constraint Satisfaction Problems, which generalizes previous approaches to building constraint-based and object-oriented configurator applications such as [12] or [33].

The innovative aspects of our work include an additional level of abstraction for constraints and nogoods, which consist of variable types instead of concrete variable instances. Furthermore, we extended the GCSP to a distributed scenario, allowing DisCSP frameworks to be adapted to dynamic configuration problems and described how this enhancement can be integrated into existing asynchronous DisCSP algorithms. The evaluation of our method on real-world-inspired distributed configuration problems of varying size and complexity revealed that the GCSP approach is advantageous compared to traditional distributed algorithms.

References

[1] A. Armstrong and E.F. Durfee, "Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems," *Proc. Int'l Joint Conf. Artificial Intelligence (IJCAI '97)*, pp. 620-625, 1997.

[2] L. Ardissono, A. Felfernig, G. Friedrich, A. Goy, D. Jannach, G. Petrone, R. Schäfer, and M. Zanker, "A Framework for the Development of Personalized, Distributed Web-Based Configuration Systems," *Artificial Intelligence Magazine*, vol. 24, no. 3, pp. 93-108, 2003.

[3] J. Amilhastre, H. Fargier, and P. Marquis, "Consistency Restoration and Explanations in Dynamic CSPs Application to Configuration," *Artificial Intelligence*, vol. 135, nos. 1/2, 199-234, 2002.

[4] H.R. Andersen, T. Hadzic, and D. Pisinger, "Interactive Cost Configuration over Decision Diagrams," *J. Artificial Intelligence Research*, vol. 37, pp. 99-140, 2010.

[5] C. Bessière, A. Maestre, and P. Meseguer, "Distributed Dynamic Backtracking," *Proc. Seventh Int'l Conf. Principles and Practice of Constraint Programming (CP '01)*, p. 772, 2001.

[6] V.E. Barker, D.E. O'Connor, J.D. Bachant, and E. Soloway, "Expert Systems for Configuration at Digital: XCON and Beyond," *Comm. ACM*, vol. 32, no. 3, pp. 298-318, 1989.

[7] H. Benameur, S. Vaucher, R. Gerín-Lajoie, P. Kropf, and B. Chaibdraa, "Towards an Agent-Based Approach for Multimarket Package E-Procurement," *Proc. Int'l Conf. Electronic Commerce Research (ICECR-5)*, 2002.

[8] C. Collin, R. Dechter, and S. Katz, "On the Feasibility of Distributed Constraint Satisfaction," *Proc. 12th Int'l Joint Conf. Artificial Intelligence (IJCAI '91)*, pp. 318-324, 1991.

[9] R. Dechter and A. Dechter, "Belief Maintenance in Dynamic Constraint Networks," *Proc. Am. Assoc. for Artificial Intelligence (AAAI '98)*, pp. 37-42, 1988.

[10] C. Domshlak, F. Rossi, K.B. Venable, and T. Walsh, "Reasoning about Soft Constraints and Conditional Preferences: Complexity Results and Approximation Techniques," *Proc. Int'l Joint Conf. Artificial Intelligence (IJCAI '03)*, pp. 215-220, 2003.

[11] F. Eizaguirre, M. Zangitu, K. Intxausti, and J. de Sosa, "A CSP Based Distributed Product Configuration System," *Proc. Workshop Configuration Systems European Conf. Artificial Intelligence (ECAI '08)*, pp. 19-22, 2008.

[12] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner, "Configuring Large Systems Using Generative Constraint Satisfaction," *IEEE Intelligent Systems and their Applications,* vol. 13, no. 4, pp. 59-68, July/Aug. 1998.

[13] A. Felfernig, G. Friedrich, and D. Jannach, "UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems," *Int'l J. Software Eng. Knowledge,* vol. 10, pp. 449-469, 2000.

[14] A. Felfernig, G. Friedrich, and D. Jannach, "Conceptual Modeling for Configuration of Mass-Customizable Products," *Artificial Intelligence Eng.,* vol. 15, no. 2, pp. 165-176, 2001.

[15] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, and M. Zanker, "Configuration Knowledge Representations for Semantic Web Applications," *Artificial Intelligence for Eng. Design, Analysis and Manufacturing,* vol. 17, pp. 31-50, 2003.

[16] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, "Consistency-Based Diagnosis of Configuration Knowledge Bases," *Artificial Intelligence,* vol. 152, pp. 213-234, 2004.

[17] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, "Towards Distributed Configuration," *Proc. Advances in Artificial Intelligence Conf. (KI '01),* vol. 2174, pp. 198-212, 2001.

[18] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, "Multi-Site Product Configuration of Telecommunication Switches," *Int'l J. Computing,* vol. 1, no. 2, 2002.

[19] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, "Semantic Configuration Web Services in the CAWICOMS Project," *Proc. First Int'l Semantic Web Conf. The Semantic Web (ISWC '02),* pp. 192-205, 2002.

[20] X. Fu and S. Li, "Multi-Ontology Based System for Distributed Configuration," *Proc. Eight Int'l Conf. Computer Supported Cooperative Work in Design,* vol. 3168, pp. 199-210, 2005.

[21] X. Fu and S. Li, "Towards a Knowledge-Based System of Distributed Configuration," *Proc. Sixth Int'l Conf. Parallel and Distributed Computing Applications and Technologies (PDCAT '05),* pp. 1024-1026, 2005.

[22] F. Frayman and S. Mittal, "Cossack: A Constraint-Based Expert System for Configuration Tasks," *Knowledge-Based Expert Systems in Eng., Planning, and Design,* D. Sriram and R.A. Adeypp, eds., pp. 143-166, Computational Mechanics, 1987.

[23] G. Gottlob, G. Greco, and T. Mancini, "Conditional Constraint Satisfaction: Logical Foundations and Complexity," *Proc. 20th Int'l Joint Conf. Artificial Intelligence (IJCAI '07),* pp. 88-93, 2007.

[24] A. García, K. O'Neill, L.J. García, P. Lord, R. Stevens, O. Corcho, and F. Gibson, "Developing Ontologies within Decentralised Settings," *Semantic e-Science, Annals of Information Systems,* vol. 11, pp. 99-139, 2010.

[25] A. Haselböck, "Exploiting Interchangeabilities in Constraint-Satisfaction Problems," *Proc. 13th Int'l Joint Conf. Artificial Intelligence (IJCAI '03),* pp. 282-289, 1993.

[26] A. Haselböck, "Knowledge-Based Configuration and Advanced Constraint Technologies," PhD thesis, Technische Universität Wien, 1993.

[27] A. Haug, "Managing Diagrammatic Models with Different Perspectives on Product Information," *J. Intelligent Manufacturing,* vol. 21, pp. 811-822, 2010.

[28] W. Havens, "Nogood Caching for Multiagent Backtrack Search," *Proc. Agents Workshop Nat'l Conf. Artificial Intelligence (AAAI '97),* 1997.

[29] M. Heinrich and W.E. Jüngst, "A Resource-Based Paradigm for the Configuring of Technical Systems from Modular Components," *Proc. IEEE Seventh Conf. Artificial Intelligence Applications (CAIA '98),* pp. 257-264, 1988.

[30] U. Junker and D. Mailharro, "Preference Programming: Advanced Problem Solving for Configuration," *Artificial Intelligence for Eng. Design, Analysis and Manufacturing,* vol. 17, no. 1, pp. 13-29, 2003.

[31] U. Junker, "Preference Programming for Configuration," *Proc. Configuration Workshop Int'l Joint Conf. Artificial Intelligence (IJCAI '01),* 2001.

[32] U. Junker, "QuickXPlain: Conflict Detection for Arbitrary Constraint Propagation Algorithms," *Proc. Modeling and Solving Problems with Constraints Workshop (IJCAI '01),* 2001.

[33] D. Mailharro, "A Classification and Constraint-Based Framework for Configuration," *Artificial Intelligence for Eng. Design, Analysis and Manufacturing,* vol. 12, no. 4, pp. 383-397, 1998.

[34] J. McDermott, "R1: A Rule-Based Configurer of Computer Systems," *Artificial Intelligence,* vol. 19, pp. 39-88, 1982.

[35] S. Mittal and F. Frayman, "Towards a Generic Model of Configuration Tasks," *Proc. 11th Int'l Joint Conf. Artificial Intelligence (IJCAI '89),* pp. 1395-1401, 1989.

[36] S. Mittal and B. Falkenhainer, "Dynamic Constraint Satisfaction Problems," *Proc. Eighth Nat'l Conf. Artificial Intelligence (AAAI),* pp. 25-32, 1990.

[37] D.L. McGuinness and J.R. Wright, "Conceptual Modelling for Configuration: A Description Logic-Based Approach," *Artificial Intelligence for Eng. Design, Analysis and Manufacturing,* vol. 12, no. 4, pp. 333-344, 1998.

[38] J.B. Pine, *Mass Customization: The New Frontier in Business Competition.* Harvard Business School Press, 1993.

[39] D. Sabin and E.C. Freuder, "Configuration as Composite Constraint Satisfaction," *Proc. AI and Manufacturing Research Planning Workshop at Nat'l Conf. Artificial Intelligence (AAAI'96),* 1996.

[40] M.-C. Silaghi and B. Faltings, "Asynchronous Aggregation and Consistency in Distributed Constraint Satisfaction," *Artificial Intelligence,* vol. 161, pp. 25-53, 2005.

[41] M. Stumptner, G. Friedrich, and A. Haselböck, "Generative Constraint-Based Configuration," *Artificial Intelligence for Eng. Design, Analysis and Manufacturing,* vol. 12, no. 4, pp. 307-320, 1998.

[42] T. Soininen, E. Gelle, and I. Niemelä, "A Fixpoint Definition of Dynamic Constraint Satisfaction," *Proc. Seventh Int'l Conf. Principles and Practice of Constraint Programming (CP '99),* pp. 419-433, 1999.

[43] M. Stumptner and A. Haselböck, "A Generative Constraint Formalism for Configuration Problems," *Proc. Third Congress of the Italian Assoc. for Artificial Intelligence on Advances in Artificial Intelligence (AI*IA '93),* pp. 302-313, 1993.

[44] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen, "Unified Configuration Knowledge Representation Using Weight Constraint Rules," *Proc. Configuration Workshop European Conf. Artificial Intelligence (ECAI '00),* pp. 79-84, 2000.

[45] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings, "Asynchronous Search with Aggregations," *Proc. 17th Nat'l Conf. Artificial Intelligence (AAAI),* pp. 917-922, 2000.

[46] M.-C. Silaghi, D. Sam-Haroud, and B.V. Faltings, "Maintaining Hierarchically Distributed Consistency," *Proc. Seventh Int'l Conf. Principles and Practice of Constraint Programming DCS Workshop (CP '00),* pp. 15-24, 2000.

[47] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings, "ABT with Asynchronous Reordering," *Proc. Intelligent Agent Technology Conf. (IAT),* pp. 54-63, Oct. 2001.

[48] M.-C. Silaghi, D. Sam-Haroud, and B.V. Faltings, "Consistency Maintenance for ABT," *Proc. Seventh Int'l Conf. Principles and Practice of Constraint Programming (CP '01),* pp. 271-285, 2001.

[49] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, "Towards a General Ontology of Configuration," *Artificial Intelligence for Eng. Design, Analysis and Manufacturing,* vol. 12, pp. 357-372, 1998.

[50] N. Staudenmayer, M. Tripsas, and C.L. Tucci, "Inter-firm Modularity and the Implications for Product Development," *J. Product Innovation Management,* vol. 22, no. 4, pp. 303-321, 2005.

[51] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara, "Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving," *Proc. 12th Int'l Conf. Distributed Computing Systems (ICDCS '92),* pp. 614-621, 1992.

[52] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara, "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms," *IEEE Trans Knowledge Data Eng.,* vol. 10, no. 5, pp. 673-685, Sept./Oct. 1998.

[53] D. Yang, M. Dong, and R. Miao, "Development of a Product Configuration System with an Ontology-Based Approach," *Computer-Aided Design,* vol. 40, no. 8, pp. 863-878, 2008.

[54] M. Yokoo and K. Hirayama, "Algorithms for Distributed Constraint Satisfaction: A Review," *Autonomous Agents and Multi-Agent Systems,* vol. 3, pp. 185-207, 2000.

[55] D. Yang, R. Miao, H. Wu, and Y. Zhou, "Product Configuration Knowledge Modeling Using Ontology Web Language," *Expert Systems with Applications,* vol. 36, pp. 4399-4411, 2009.

[56] M. Yokoo, "Asynchronous Weak-Commitment Search for Solving Large-Scale Distributed Constraint Satisfaction Problems," *Proc. Int'l Conf. Multi-Agent Systems (ICMAS '95),* pp. 467-318, 1995.

[57] M. Zanker, D. Jannach, M.-C. Silaghi, and G. Friedrich, "A Distributed Generative CSP Framework for Multi-Site Product Configuration," *Proc. 12th Int'l Workshop Cooperative Information Agents,* pp. 131-146, 2008.

Dietmar Jannach is a full professor and head of the e-services research group at TU Dortmund, Germany. His general research interests lie in the application of intelligent systems technology in practice. His recent research focus is on product configurators as well as interactive advisory and recommender systems.

Markus Zanker is an associate professor at Alpen-Adria Universität Klagenfurt, Austria. He is the author of numerous scientific papers in the area of knowledge-based systems, product configuration and recommender systems and also a cofounder and director of ConfigWorks GmbH, a provider of interactive selling solutions.