Chapter 18

Distributed Generative Constraint Satisfaction

They that can give up essential liberty to obtain a little temporary safety
deserve neither liberty nor safety.
Benjamin Franklin

In the previous chapter we have analyzed our initially motivating application for distributed search with private problems. Due to the generality of constraint satisfaction, other applications can also benefit from these techniques. With Markus Zanker and his colleagues from TU-Klagenfurth, we have recently studied an extension of DisCSPs for modeling distributed configuration problems. In the following we will review the obtained framework and its new requirements from search protocols.

18.1 Problem

Today's configurators are centralized systems and do not allow manufacturers to cooperate online for offer-generation or sales-configuration. However, supply chain integration of configurable products requires the cooperation of the configuration systems from the different manufacturers that jointly offer solutions to customers. As a consequence, there is a high potential for methods that enable the computation of such configurations by independent specialized agents. Several approaches to *centralized* configuration tasks are based on constraint satisfaction problem (CSP) solving. Most of them extend the traditional CSP approach in order to comply to the specific expressivity and dynamism requirements for configuration and synthesis tasks.

The distributed generative CSP (DisGCSP) framework proposed here builds on a CSP formalism that encompasses the *generative* aspect of variable creation and extensible domains of problem variables. It also builds on the distributed CSP (DisCSP) framework, allowing for approaches to configuration tasks where the knowledge is distributed over a set of agents. Notably, the notions of constraint and nogood are generalized to an additional level of abstraction, extending inferences to types of variables. The usage of the new framework is exemplified by describing modifications to some complete algorithms for DisCSP when targeting DisGCSPs.

The paradigm of mass-customization allows customers to tailor (configure) a product or service according to their specific needs, i.e. the customer can select between several features and options that should be included in the configured product and can determine the physical component structure of the personalized product variant. Typically, there are several technical and marketing restrictions on the legal parameter constellations and the physical layout. This led manufacturers to develop support for checking the feasibility of user requirements and for computing a consistent solution. This functionality is provided by product configuration systems (configurators), whereby they have shown to be a successful application area for different AI techniques (Stumptner 1997) such as description logics (McGuiness & Wright 1998), or rule-based (Barker *et al.* 1989) and

constraint-based solving algorithms. (Fleischanderl *et al.* 1998) describes the industrial use of constraint techniques for the configuration of large and complex systems such as telecommunication switches and (Mailharro 1998) is an example of a powerful tool based on Constraint Satisfaction available on the market.

However, companies find themselves in dynamically determined coalitions with other highly specialized solution providers that jointly offer customized solutions. This high integration aspect of todays digital market implies that software systems supporting the selling and configuration task must no longer be conceived as standalone systems. A product configurator can be therefore seen as an agent with private knowledge that acts on behalf of its company and cooperates with other agents to solve a configuration task. This chapter abstracts the *centralized* definition of a configuration task in (Stumptner *et al.* 1998) to a more general definition of a *generative* CSP that is also applicable to the wider range of synthesis problems. Furthermore, we propose a framework that allows to address distributed configuration tasks by extending DisCSPs with the innovative aspects of local generative CSPs:

1. The constraints (and nogoods) are generalized to a form where they can depend on types rather than on identities of variables. This also enables an elegant treatment of the next aspects.

2. The number of variables of certain types that are active in the local CSP of an agent, may vary depending on the state of the search process. In the DisCSP framework, the external variables existing in the system are predetermined, but here the set of variables defining the problem is determined dynamically.

3. The domain of the variables may vary dynamically. Some variables model possible connections and they depend on the existence of components that could become connected.

We also describe the interesting impact of the previously mentioned changes on asynchronous algorithms. In the following we motivate our approach with an example, Section 3 defines a *generative* CSP and in Section 4 distributed generative CSP is formalized and extensions to current DisCSP frameworks are presented.

18.2 Motivating example

Now we will see as example the well known N-queens problem, that has some similar properties with configuration problems. The characteristics of a distributed configuration problem or similar distributed synthesis tasks are integrated into our N-queens scenario:

(a) parts of the problem (i.e., constraints) are shared among agents and

(b) the problem is dynamically extended (i.e., N is increased), if no solution can be found.

Adding additional problem variables leads to domain extensions and thus to a larger search- and solution space. The goal is to place N queens on distinct squares in an $N \times N$ chess board, where no two queens threaten each other (Tsang 1993). We formalize the problem by making each row of the board a problem variable x_i, where the subscript i ensures unique variable names. In a distributed setting we employ three agents, each owning a fraction of the constraints necessary to solve the N-queens problem. Furthermore, we want to show the *generative* aspect of problem solving in the example, where agents start with a representation of a 0-queens problem and specific requirements on the final solution possibly imposed by the customer. Once the agents determine that a solution cannot be found, they extend the problem space by adding an additional row which in consequence enlarges the domain of row variables by one. Since the exact number of problem variables is not known from the beginning, constraints cannot be directly formulated on concrete variables. Instead, comparable to programming languages, variable types exist that allow to associate a newly created variable with a domain and we can specify relationships in terms of *generic constraints*. (Stumptner *et al.* 1998) define a generic constraint γ as a constraint schema,

where meta-variables V^t act as placeholders for concrete variables of a specific type t[1]. In our example three types of problem variables exist, representing the even (t_e) and the uneven rows (t_u) as well as a type (t_c) of counter variables for the number of instantiations of each type (x_{type}), which allows us to distribute the N-queens constraints among the agents. Therefore, each agent a_i possesses a set of private constraints Γ^{a_i}, i.e., $\Gamma^{a_1} = \{\gamma_1, \gamma_2, \gamma_3, \gamma_8, \gamma_9\}$, $\Gamma^{a_2} = \{\gamma_4, \gamma_5, \gamma_8, \gamma_9\}$ and $\Gamma^{a_3} = \{\gamma_6, \gamma_7, \gamma_8, \gamma_9\}$, that are defined as follows:

- $\gamma_1 : val(x_{t_u}) = val(x_{t_e}) \vee val(x_{t_u}) = val(x_{t_e}) + 1$, where $val(x)$ is a predicate that gives the assigned value of variable x.

  Informally, the number of uneven rows may exceed the number of even rows by one.

- $\gamma_2 : val(V^{t_u}) \neq val(V^{t_e})$

  Informally, no two queens on an even and an uneven row are allowed to take the same column value.

- $\gamma_3 : abs(2 \times (index(V^{t_u}) - index(V^{t_e})) - 1) \neq abs(val(V^{t_u}) - val(V^{t_e}))$, where $index(x)$ returns a number i indicating that x is the i^{th} variable of its type and $abs(n)$ is a predicate that returns the absolute of n.

  No two queens on an even and an uneven row are allowed to be on the same diagonal.

- $\gamma_4 : V_1^{t_u} \neq V_2^{t_u} \rightarrow val(V_1^{t_u}) \neq val(V_2^{t_u})$.

  No two queens on uneven rows are allowed to take the same column value.

- $\gamma_5 : V_1^{t_u} \neq V_2^{t_u} \rightarrow abs(2 \times (index(V_1^{t_u}) - index(V_2^{t_u}))) \neq abs(val(V_1^{t_u}) - val(V_2^{t_u}))$.

  No two queens on uneven rows are allowed to be on the same diagonal.

- $\gamma_6 : V_1^{t_e} \neq V_2^{t_e} \rightarrow val(V_1^{t_e}) \neq val(V_2^{t_e})$.

  No two queens on even rows are allowed to take the same column value.

- $\gamma_7 : V_1^{t_e} \neq V_2^{t_e} \rightarrow abs(2 \times (index(V_1^{t_e}) - index(V_2^{t_e})) \neq abs(val(V_1^{t_e}) - val(V_2^{t_e}))$.

  No two queens on even rows are allowed to be on the same diagonal.

- $\gamma_8 : val(V^{t_u}) \leq x_{t_e} + x_{t_u}$. $\gamma_9 : val(V^{t_e}) \leq x_{t_e} + x_{t_u}$.

  The latter two constraints delimit the domain of row variables to the total number of rows.

The Figure 18.1 depicts the initial situation, with a 0-queens problem. The customer requests agent a_1 to satisfy the requirement of finding a solution containing at least two uneven rows:

$$\gamma_{cust} : x_{t_u} \geq 2.$$

Having added γ_{cust} to the set of private constraints of agent a_1, the search process starts and the solution space is continuously extended by the instantiation of additional problem variables, until a solution is found for a 4-queens problem that satisfies all local constraints of the agents. The links between two agents indicate that they share variables, which is described in more detail later on. Thus, a solution to a generative constraint satisfaction problem requires not only finding valid assignments to variables, but also determining the exact size of the problem itself. In the sequel of the chapter we define a model for the local configurators and we detail extensions to DisCSP algorithms.

[1]The exact semantics of generic constraints is given in Definition 18.2 in Section 18.3.

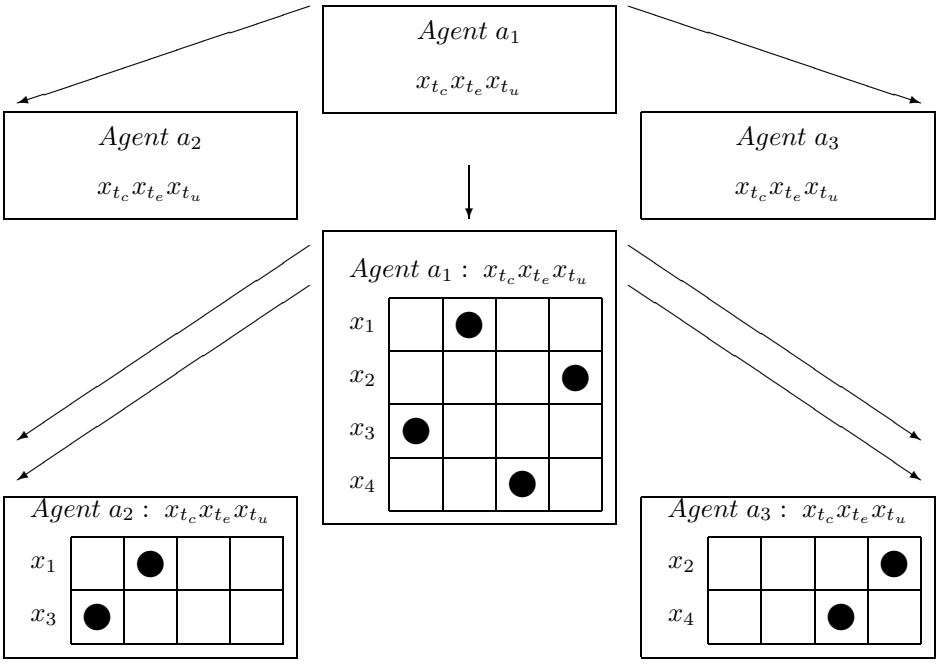

Figure 18.1: Motivating example.

18.3 Generative Constraint Satisfaction

Conventional CSP formulations are static. They consist of a fixed set of variables having a domain and a set of constraints on these variables. However in many applications, solving is a *generative* process, where the number of involved components is not known from the beginning. To represent these problems we employ an extended formalism that complies to the specifics of configuration and other synthesis tasks where problem variables representing components of the final system are *generated* dynamically as part of the solution process because their total number cannot be determined beforehand. The framework is called *generative* CSP (GCSP)(Haselböck 1993b; Stumptner *et al.* 1998; Nareyek 2001). This kind of dynamism extends the approach of dynamic CSP (DCSP) formalized by Mittal and Falkenhainer (Mittal & Falkenhainer 1990), where all possibly involved variables are known from the beginning. This is needed because the activation constraints reason on the variable's activity state. (Sabin & Freuder 1996) propose a conditional CSP to model a configuration task, where structural dependencies in the configuration model are exploited to trigger the activation of subproblems. Another class of DCSP was first introduced by (Dechter & Dechter 1988) where constraints can be added or removed independently of the initial problem statement. The dynamism occurring in a GCSP differentiates from the one described in (Dechter & Dechter 1988) in the sense that a GCSP is extended in order to find a consistent solution and the latter has already a solution and is extended due to influence from the outside world (e.g., additional constraints) that necessitates finding a new solution. Here we give a definition of a GCSP that abstracts from the configuration task specific formulation in (Stumptner *et al.* 1998) and applies to the wider range of synthesis problems.

Definition 18.1 (Generative constraint satisfaction problem (GCSP)) *A generative constraint satisfaction problem is 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 $t_i \in T$, a counter variable $x_{t_i} \in X_0$ 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 $X \times (T, \mathbb{N}_+)$, where $\mathbb{N}_+$ is the set of positive integer numbers. Each tuple $(x, (t, i))$ associates a variable $x \in X$ with a unique type $t \in T$ and an index i, that indicates x is the i^{th} variable of type t. The function $type(x)$ accesses Δ and returns the type $t \in T$ for x and the function $index(x)$ returns the index of x.*

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

When modeling a configuration problem, variables representing named connection points between components, i.e., *ports*, will have references to other ports as their domain. Consequently, we need variables whose domain varies depending on the size of a set of specific variables (Stumptner *et al.* 1998).

Example 18.27 *Given t_{mod} as the type of variables representing modules and t_{port} as the type of port variables that are allowed to connect to modules, then the domain of the port variables $dom(t_{port})$ must contain references to modules. This is specified by defining $dom(t_{port}) = \{1, \ldots, ub\}$, where ub is an upperbound on the number of variables of type t_{mod}, and formulating an additional generic constraint that restricts all variables of type t_{port} using the counter variable for the total number of variables having type t_{mod}, i.e., $val(V^{t_{port}}) \leq x_{t_{mod}}$. With the help of the index() function concrete variables can then be referenced.*

Referring to our introductory example we can formalize the local GCSP of agent a_1 (initially consisting only of counter variables x_{t_i}, their type t_c, and the types of row variables) as $X^{a_1} = \{x_{t_c}, x_{t_e}, x_{t_u}\}$, $\Gamma^{a_1} = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5\}$, $T^{a_1} = \{t_c, t_e, t_u\}$ and $\Delta^{a_1} = \{(x_{t_c},(t_c, 1)),(x_{t_e},(t_c, 2)), (x_{t_u},(t_c, 3))\}$. The domain for even and uneven row variables is consequently defined as $dom(t_e) = dom(t_u) = dom(t_c) = \{1, \ldots, ub\}$, where the domains for the row variables are limited by the domain constraints (i.e., γ_8, γ_9).

Definition 18.2 (Generic constraint) *A generic constraint $\gamma \in \Gamma$ formulates a restriction on the meta-variables $M_a, \ldots, M_k$. A meta-variable M_i is associated a variable type $type(M_i) \in T$ and must be interpreted as a placeholder for all concrete variables x_j, where $type(x_j) = type(M_i)$.*

Note that generic constraints can also formulate restrictions on specific initial variables from X_0 by employing the *index*() function.

Consider the GCSP(X, Γ, T, Δ) and let $\gamma \in \Gamma$ restrict the meta-variables $M_a, \ldots, M_k$, where $type(M_i) \in T$ is the defined variable type of the meta variable M_i.

Definition 18.3 (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 meta-variable M_i is substituted by the concrete variable x_i.

Thus a *generic* constraint must be seen as a constraint scheme that is expanded into a set of constraints after a preprocessing step, where meta-variables are replaced by all possible combinations of concrete variables having the same type, e.g., given a GCSP of agent a_1 (excluding counter variables) with $X^{a_1} = \{x_1, x_2, x_3\}$, $T^{a_1} = \{t_u, t_e\}$ and $\Delta^{a_1} = \{(x_1, (t_u, 1)), (x_2, (t_e, 1)), (x_3, (t_u, 2))\}$, the satisfiability of the generic constraint γ_2 is checked by testing the following conditions: $val(x_1) \neq val(x_2)$. $val(x_3) \neq val(x_2)$.

Definition 18.4 (Solution for a generative CSP) *Given a generative constraint satisfaction problem GCSP(X_0, Γ, T, Δ_0), then its solution encompasses the finding of a set of variables X, type and index assignments Δ and an assignment tuple θ for the variables in X, s.t.*

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

We do not impose a minimality criteria on the number of variables in our solution, because in practical applications different optimization criteria exist, such as total cost or flexibility of the solution, thus non-minimal solutions can be preferred over minimal ones.

The calculated solution (excluding counter variables) for the local GCSP of agent a_1 consists of $X^{a_1} = \{x_1, x_2, x_3, x_4\}$, $\Delta^{a_1} = \{(x_1, (t_u, 1)),(x_2, (t_e, 1)),(x_3, (t_u, 2)),(x_4, (t_e, 2))\}$ and the assignment tuple $x_1 = 2$, $x_2 = 4$, $x_3 = 1$ and $x_4 = 3$. Thus, $x_1, \ldots, x_4$ are the names of *generated* variables.

Names for generated variables are unique and can be randomly chosen by the GCSP solver implementation and therefore constraints must not formulate restrictions on the variable names of generated variables. Thus substitution of any generated variable (i.e., $X \setminus X_0$) by a newly generated variable with equal type, index and value assignment has no effect on the consistency of generic constraints.

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

18.4 Framework for DisGCSP

We show how the DisCSP framework can be also applied to a scenario where each agent has to locally solve a generative constraint satisfaction task. Each time an agent extends the solution space of his local GCSP by creating an additional variable, the DisCSP setting is transformed into a new DisCSP setting, which again has all properties required by asynchronous search to correctly function.

In order to solve a distributed configuration problem the GCSP approach has to be extended to a multi-agent scenario, where each agent wants to satisfy a local GCSP and agents keep their constraints private for security and privacy reasons, but share all variables which they are interested in. As constraints employ meta-variables, the *interest* of an agent in variables needs to be redefined:

Definition 18.5 (Interest in variables) *An agent a_j owning a local $GCSP^{a_j}(X^{a_j},\Gamma^{a_j},T^{a_j},\Delta^{a_j})$ is said to be interested in a variable $x \in X^{a_h}$ of an agent a_h, if there exists a generic constraint $\gamma \in \Gamma^{a_j}$ formulating a restriction on the meta-variables $M_a, \ldots, M_k$, where $type(M_i) \in T^{a_j}$ is the defined variable type of the meta variable M_i, and $\exists M_i \in M_a, \ldots, M_k : type(x) = type(M_i)$.*

Definition 18.6 (Distributed generative CSP) *A distributed generative constraint satisfaction problem has the following characteristics:*

- *$A = \{a_1, \ldots, a_n\}$ is a set of n agents, whereby each agent a_i owns a local $GCSP^{a_i}(X^{a_i},\Gamma^{a_i},T^{a_i},\Delta^{a_i})$.*

- *All variables in* $\bigcup_{i=1}^{n} X^{a_i}$ *and all type denominators in* $\bigcup_{i=1}^{n} T^{a_i}$ *share a common name-space, ensuring that a symbol denotes the same variable, resp. the same type, with every agent.*
- *For every pair of agents* $a_i, a_j \in A$ *and for every variable* $x \in X^{a_j}$*, where agent* a_i *is interested in* x*, must hold* $x \in X^{a_i}$*.*
- *For every pair of agents* $a_i, a_j \in A$ *and for every shared variable* $x \in X^{a_i} \cap X^{a_j}$ *the same type and index must be associated to* x *in the local GCSPs of the agents, i.e.,* $type^{a_i}(x) = type^{a_j}(x) \wedge index^{a_i}(x) = index^{a_j}(x)$*.*

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

Given a distributed generative constraint satisfaction problem among a set of n agents then its solution encompasses an assignment tuple θ for every variable $x \in \bigcup_{i=1}^{n} X^{a_i}$, where $\theta = \bigcup_{i=1}^{n} \theta^{a_i}$ and θ^{a_i} is a solution for the local $GCSP_i^a$ of agent a_i.

Definition 18.7 (Generic aggregate) *A generic aggregate is a unary generic constraint. It takes the form:* $\langle M, i, s, h \rangle$*, where* M *is a meta-variable,* i *is a set of index values for which the constraint applies,* s *is a set of values, and* h *is a signature of the aggregate.*

Definition 18.8 (Generic nogood) *A generic nogood takes the form* $\neg N$*, where* N *is a set of generic aggregates for distinct meta-variables.*

Given the characteristics of a DisGCSP (see Definition 18.4) the links can be initialized before the start of the algorithm, due to the common naming space for type denominators and the condition of a unique type and index assignment to variables over all agents. Value assignments to variables are communicated to agents via **ok?** messages that transport *generic aggregates* in our DisGCSP framework, which represent domain restrictions on variables by unary constraints. Each of these unary constraints in our DisGCSP has attached an unique identifier called constraint reference (CR) (Silaghi *et al.* 2000g). Any inference has to attach the CRs associated to arguments into the obtained nogood. We treat the extension of the domains of the variables as a constraint relaxation (Silaghi *et al.* 2000g). For this reason we introduce next features for algorithm extensions:

- **announce** messages broadcasts a tuple (x, t, i), where x is a newly created variable of type t and with index i to all other agents. The receiving agents determine their interest in variable x and react in one of the following ways (a) send an $addlink(x)$ message (b) add the sending agent to its outgoing links or (c) discard the message. They also may broadcast **domain** messages.
- **domain** messages broadcasts a set CR of obsolete constraint references. Any receiving agent removes all the nogoods having attached to them a constraint reference $cr \in CR$. The receiver of the message calls then the function *check_agent_view()* detailed in (Yokoo *et al.* 1992), making sure that it has a consistent proposal or that it generates nogoods.
- **nogood** messages transport *generic nogoods* $\neg N$ that contain assignments for meta-variable instances. These messages are multicast to all agents interested in $\neg N$. An agent A_i is interested in a generic nogood $\neg N$ if any meta-variable in $\neg N$ interests either A_i or agents with higher priority than A_i, and at least one of them is interesting for A_i.
- When an agent needs to revoke the creation of a new variable due to backtracking in his local solving algorithm, he assigns it a specific value from its domain indicating the deactivation of the variable and communicates it via an **ok?** to all interested agents.

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

Theorem 18.1 *Whenever an existing extension of AAS is extended with the previous messages and is applied to DisGCSPs, the obtained protocols are correct, complete and terminate.*

Proof. Let us consider that we extend a protocol called P.

Completeness: All the generated information results by inference. If failure is inferred (when no new component is available), then indeed no solution exists.

Termination: Without introducing new variables, the algorithm terminates. Since the number of variables that can be generated is finite, termination is ensured.

Correctness: The resulting overall protocol is an instance of P, where the delays of the system agent initializing the search equals the time needed to insert all the variables generated before termination. Therefore the result satisfies all the agents and the solution is correct. □

To help readers get an intuition of the framework, we also mention that besides typical parameters for DisCSP generators (Hirayama *et al.* 2000; Silaghi *et al.* 2000g), a random DisGCSPs generator for benchmarking has the additional parameters (a) number of added variables per failure, (b) a random distribution for the types of new variables and (c) number of added values per added variable per existing variable.

18.5 Conclusions

Building on the definition of a centralized configuration task from (Stumptner *et al.* 1998), we formally defined a new class of CSP, termed generative CSP (GCSP), that generalizes the approaches of constraint-based configurator applications in use (Fleischanderl *et al.* 1998; Mailharro 1998). The innovative aspects include generic constraints, dynamic variable creation and extensible domains. Furthermore, we extended GCSP to a scenario, where knowledge is distributed among several agents and proposed modifications to complete algorithms for DisCSP solving.

We have proposed in this chapter an additional level of abstraction for constraints and nogoods. Constraints and nogoods can refer to types of variables. This abstraction adapts well DisCSP frameworks for dynamic configuration problems (but it can be used in static models as well). We have described how this enhancement can be naturally integrated in a large family of existing asynchronous algorithms for DisCSPs.