A Fixpoint Definition of Dynamic Constraint Satisfaction

Timo Soininen[1], Esther Gelle[2], and Ilkka Niemelä[3]

[1] Helsinki University of Technology, TAI Research Center,
P.O.Box 9555, FIN-02015 HUT, Finland
Timo.Soininen@hut.fi
[2] ABB Corporate Research Ltd,
CHCRC.C2 Segelhof, CH-5405 Baden, Switzerland
Esther.Gelle@ch.abb.com
[3] Helsinki University of Technology, Dept. of Computer Science and Eng.,
Laboratory for Theoretical Computer Science,
P.O.Box 5400, FIN-02015 HUT, Finland
Ilkka.Niemela@hut.fi

Abstract. Many combinatorial problems can be represented naturally as constraint satisfaction problems (CSP). However, in some domains the set of variables in a solution should change dynamically on the basis of assignments of values to variables. In this paper we argue that such *dynamic constraint satisfaction problems (DCSP)*, introduced by Mittal and Falkenhainer, are more expressive than CSP in a knowledge representation sense. We then study the problem of generalizing the original DCSP with *disjunctive activity constraints* and *default negation* which are useful in, e.g., product configuration problems. The generalization is based on a novel definition of a solution to DCSP. It uses a fixpoint condition instead of the subset minimality condition in the original formulation. Our approach coincides with the original definition when disjunctions and default negations are not allowed. However, it leads to lower computational complexity than if the original definition were generalized similarly. In fact we show that the generalized DCSP is **NP**-complete. As a proof of concept, we briefly describe two novel implementations of the original DCSP and give test results for them.

1 Introduction

Constraint satisfaction problems (CSP) provide a convenient framework for representing combinatorial tasks. Powerful search algorithms have also been developed for solving CSPs [13] by integrating filtering (consistency algorithms) into a search algorithm. A CSP consists of a set of variables with domains on which allowed value combinations are specified as constraints. However, e.g. product configuration [8] problems exhibit dynamic aspects in the generation of problem spaces, i.e., the set of variables in a solution changes dynamically on the basis of assignments of values to variables. When configuring a mixer, for example, a condenser is a typical optional component which does not have to be present in every solution. It is only necessary if the vessel volume is large and chemical reactions occur during the mixing process.

Such dynamic aspects are difficult to capture in a standard CSP in which all variables are assigned values in every solution. One way to deal with an optional component is to add a special NULL value to the domain of the variable representing the component [3]. In addition, each constraint that refers to the variable needs to be modified to function properly in the presence of NULL. More seriously, an additional constraint is needed to prevent other values than NULL for the variable if there is no reason for including the optional component in the solution. Such constraints may have a very large arity and cardinality, as many variables may affect the value of the variable representing the optional component. Much effort has therefore been put to include dynamic aspects in CSPs [5, 7, 12, 3]. One of these is the framework of *dynamic constraint satisfaction problems (DCSP)* [5] which adds *activity constraints* to CSP. The activity constraints govern which variables are given values, i.e. are *active*, in a solution.

In this paper we provide evidence that there is a significant difference in the expressivity of DCSP and CSP in a knowledge representation sense using a concept of modularity [10]. Modularity of representation means that a small change of knowledge results in a small change in its representation. It is shown that a DCSP cannot be modularly represented as a CSP. Despite this increased expressiveness, the activity constraints of DCSP are relatively limited for product configuration problems. It is, e.g., difficult to encode, as an activity constraint, that for chemical dispersion either a condenser or a cooler should be included in the configuration of a mixer. Representing this type of knowledge conveniently requires *disjunctive activity constraints* [5]. However, already Mittal and Falkenhainer [5] note that finding solutions to DCSPs with disjunctive activity constraints seems to be computationally very expensive. We confirm this observation by showing that extending the original definition results easily in $\Sigma_2\mathbf{P}$-hard decision problems. This is due to a subset minimality condition on solutions in the original definition.

In order to keep the computational complexity of the decision problems for DCSP in $\mathbf{NP}$, we present a new definition in which the minimality condition is replaced with a fixpoint condition. The definition allows more expressive activity constraints generalized to include disjunctions and *default negation* on constraints and their activity. The default negation is handled similarly as in the stable model semantics of normal logic programs [2]. Our definition of DCSP coincides with the original one when disjunctive activity constraints and default negations are not allowed. It does not guarantee subset minimality of solutions when disjunctions are permitted but ensures that active variables are justified in a weaker and computationally much more feasible sense. We feel that subset minimality is in fact a specific optimality criterion which is not very relevant in many applications where measuring, e.g., cost and resource consumption are more important. Our idea is that such optimality criteria can be added on top of our definition whose role is to provide the solutions satisfying the constraints.

Our generalization does not increase the complexity of the relevant decision tasks, which are shown to be **NP**-complete. This also provides a complexity result for the original DCSP class. As DCSP and CSP are computationally equally complex and DCSP is more expressive than CSP, DCSP seems to be a more feasible framework than CSP for domains with dynamic aspects.

We briefly discuss two novel implementations of the original DCSP to give a proof of concept. The first is based on an extension of a basic CSP algorithm, while the other is based on translating a DCSP to a set of rules in a logic program-like language [10]. The implementations exhibit acceptable running times for several problems from the configuration domain. We finally present some conclusions and topics for future work.

2 The DCSP Formalism

We first recall the original definition of a dynamic constraint satisfaction problem. An instance $\mathcal{P}$ of DCSP is of the form $\langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$, where $\mathcal{V} = \{v_1, \ldots, v_n\}$ is the set of *variables* and $\mathcal{D} = \{D_1, \ldots, D_n\}$ is the set of *domains* of the variables providing a set $D_i = \{d_{i1}, \ldots, d_{ij}\}$ of *values* for each variable v_i. The set of *initial* variables of $\mathcal{P}$ is denoted by $\mathcal{V}_I$, $\mathcal{V}_I \subseteq \mathcal{V}$, the set of *compatibility constraints* by $\mathcal{C}_C$ and the set of *activity constraints* by $\mathcal{C}_A$. We assume that all these sets are finite. Next, we define a *legal assignment* to a set of variables.

Definition 1. *An assignment of a value d_{ij} to a variable v_i is of the form $v_i = d_{ij}$, where $d_{ij} \in D_i$. A legal assignment $\mathcal{A}$ to a DCSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ is a set of assignments with at most one assignment for each variable.*

In contrast to CSP, a variable can be in one of two states. A variable is *active* iff it is assigned a value in an assignment, otherwise it is *not active*.

A compatibility constraint c with arity j specifies the set of allowed combinations of values for a set of variables $v_1, \ldots, v_j$ as a subset of the Cartesian product of the domains of the variables. We denote the subset by $c(v_1, \ldots, v_j)$, i.e. $c(v_1, \ldots, v_j) \subseteq D_1 \times \ldots \times D_j$. A compatibility constraint is *active* iff all the variables it constrains are active. An activity constraint that activates variable v is of form $c \overset{ACT}{\rightarrow} v$, where c is defined equivalently to a compatibility constraint. Here we generalize the original definition slightly in that we do not require that the activated variable is distinct from the variables that the constraint on the left hand side refers to. We use the notation $v_i = d_{ij} \overset{ACT}{\rightarrow} v_k$ for activity constraints whose left hand side consists of one constraint with one tuple, and sometimes enclose the entire activity constraint in parentheses for clarity. *Always require*, *require not* and *always require not* activity constraints [5] are treated as shorthand notation for simplicity.[1]

[1] For example, an always require constraint of the form $v_1, \ldots, v_i \overset{ACT}{\rightarrow} v_j$ is defined as $c \overset{ACT}{\rightarrow} v_j$ where c allows all value combinations on the variables $v_1, \ldots, v_i$, i.e $c = D_1 \times \ldots \times D_i$.

A solution to a DCSP is a subset minimal legal assignment that satisfies all compatibility and activity constraints and has assignments for the initial variables. The subset minimality condition in effect excludes assignments with active variables whose activity is not *justified* by the variables being in the set of initial variables or by at least one activity constraint.

Definition 2. *A legal assignment $\mathcal{A}$ for a DCSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ satisfies the constraints in $\mathcal{P}$ iff the following conditions hold:*

1. *$\mathcal{A}$ satisfies each compatibility constraint $c \in \mathcal{C}_C$, i.e., if c is active in $\mathcal{A}$ then the variables constrained by c are assigned values allowed by c.*
2. *$\mathcal{A}$ satisfies each activity constraint $(c \overset{ACT}{\rightarrow} v_k) \in \mathcal{C}_A$,, i.e., if c is active in and satisfied by $\mathcal{A}$ then v_k is active in $\mathcal{A}$.*

Definition 3. *A legal assignment $\mathcal{A}$ is a solution to a DCSP iff $\mathcal{A}$*

1. *satisfies the constraints,*
2. *contains assignments of values to initial variables,*
3. *is subset minimal, i.e. there is no assignment $\mathcal{A}_1$ satisfying 1. and 2. such that $\mathcal{A}_1 \subset \mathcal{A}$.*

Example 1. We use the simplified configuration task of an industrial mixer [14] to illustrate a DCSP. A *product configuration task* takes as input a *configuration model* and a set of *requirements*. The configuration model describes all the components that can be included in a product and all the dependencies that define how components can be combined. The requirements specify additional constraints on the product individual to be configured. The output of the configuration task is a *legal configuration* that satisfies the requirements. A *legal configuration* is a configuration that is allowed by the configuration model, i.e., that contains only components defined by the model and that satisfies the dependencies in it. In the framework of DCSP, the configuration model is described as a DCSP the solutions to which correspond to the set of legal configurations. The requirements are represented as a set of activity and compatibility constraints.

An industrial mixer can be used for different mixing processes, e.g. chemical reactions, mixing of side products etc. It consists of a set of standard components, such as a vessel containing the products to be mixed, the mixer itself with impellers and an engine, etc. Depending on the chemical properties of the products to be mixed, heat is produced which requires the use of a cooler or a condenser. To represent the configuration model of a mixer as a DCSP, the components and their properties, for example the volume, are represented as variables. The mixing process is represented by a variable for the mixing task. The components can be of different types, e.g. the mixer can be a reactor, storage tank, or simple mixer. These as well as the different types of mixing tasks are represented as discrete domains of the variables.[2] As a cooler and a condenser

[2] To designate variables and values in the problem, we use the first letters of their names.

are not needed in every solution, they are not included in the set of initial variables but are introduced by activity constraints. The configuration model for the mixer configuration task consists of the variables in Table 1 and the activity constraints $\mathcal{C}_A = \{a_1, a_2\}$ and compatibility constraints $\mathcal{C}_C = \{c_1, c_2\}$ given as follows:

$$\begin{aligned}
a_1 &= (Mi = r \overset{ACT}{\rightarrow} Coo)\\
a_2 &= (Mt = d \overset{ACT}{\rightarrow} Con)\\
c_1(Con, Vol) &= \{(con1, l), (con2, l), (con2, s)\}\\
c_2(Mi, Vol) &= \{(r, s), (m, s), (m, l), (t, s), (t, l)\}
\end{aligned}$$

Table 1. Variables of the mixer configuration model.

Variable	**Description**	**Domain**
Mt $\in \mathcal{V}_I$	Mixing task	{d(*dispersion*),s(*suspension*),b(*blending*)}
Mi $\in \mathcal{V}_I$	Mixer	{m(*mixer*),r(*reactor*),t(*tank*)}
Coo	Cooler	{coo1(*cooler1*)}
Con	Condenser	{con1(*condenser1*),con2(*condenser2*)}
Vol $\in \mathcal{V}_I$	Volume	{l(*large*),s(*small*)}

Given the above mixer DCSP, the assignment $\mathcal{A}_1 = \{Mi = m, Mt = b, Vol = l\}$ is a solution, i.e. a legal configuration, since it satisfies the constraints, contains assignments for the initial variables and there is no other solution $\mathcal{A}$ such that $\mathcal{A} \subset \mathcal{A}_1$. The assignment $\mathcal{A}_2 = \{Mi = r, Mt = b, Vol = s, Coo = coo1\}$ is another solution with a different set of active variables. However, $\mathcal{A}_3 = \{Mi = r, Mt = b, Vol = s, Coo = coo1, Con = con2\}$ is not a solution since the assignment $Con = con2$ is not justified by Con being in the set of initial variables or by being activated by an activity constraint, and thus $\mathcal{A}_2 \subset \mathcal{A}_3$.

The basic task in the DCSP framework is to find a solution to a given DCSP. However, for the configuration domain the relevant task is to find a solution that satisfies a set of requirements, which may also refer to the activity of variables. This is important when customer requirements concerning some functionality lead to conditions on the activity of the corresponding variables. We therefore analyze in subsequent sections the computational complexity of the decision versions of these tasks, defined as follows:

Definition 4. *$DCSP(D)$: Given a DCSP $\mathcal{P}$, is there a solution for $\mathcal{P}$?*
$DCSP_R(D)$: Given a DCSP $\mathcal{P}$ and a set of requirements R, is there a solution S for $\mathcal{P}$ such that S satisfies R?

Note that the requirements of the configuration task cannot be handled by simply adding constraints or initial variables to a DCSP representing the configuration model, as this can change the configuration model to allow configurations which were not originally intended. For example, consider the following subset of the mixer problem:

Example 2. The configuration model is defined as follows: $\mathcal{V} = \{Mi, Coo\}$, the variables have the same domains as previously, $\mathcal{V}_I = \{Mi\}$, $\mathcal{C}_C = \emptyset$, and $\mathcal{C}_A = \{(Mi = r \overset{ACT}{\rightarrow} Coo)\}$. The set of legal configurations is now $\{\{Mi = r, Coo = cool\}, \{Mi = t\}, \{Mi = m\}\}$. Consider the set of requirements $\{(\overset{ACT}{\rightarrow} Coo), c(Mi) = \{(m)\}\}$, i.e., that cooler is active and $Mi = m$. There is no legal configuration with respect to the configuration model which would satisfy these requirements. However, adding them to the configuration model would change the set of legal configurations to $\{\{Mi = m, Coo = cool\}\}$, permitting a configuration that satisfies the requirements but is not allowed by the original configuration model.

3 Expressivity of DCSP vs. CSP

In this section we show that DCSP is more expressive than CSP in a knowledge representation sense. We use the concept of *modularity* [10] to establish this. Intuitively, a modular representation of knowledge is such that a small change in the knowledge leads to a small change in its representation. This property is important for maintaining and updating the knowledge. For example, if a DCSP could be represented as a CSP in a modular fashion, a simple update like adding a variable to the set of initial variables should result in a local change to the corresponding CSP. It seems that there is no modular representation of a DCSP as a CSP even under very weak notions of modularity. Here we demonstrate this in the case of updating the set of initial variables. This result can be extended to involve more complex changes to DCSP, such as additions and removals of compatibility and activity constraints and their allowed value tuples.

More precisely, we say that DCSP is *modularly representable* by CSP iff there is a mapping for each DCSP $\mathcal{P}$ to a CSP $\mathrm{T}(\mathcal{P})$ such that the solutions to $\mathcal{P}$ *agree* with the solutions to $\mathrm{T}(\mathcal{P})$ and the effect of changing the set of initial variables in $\mathcal{P}$ can be accomplished by a *simple update* to $\mathrm{T}(\mathcal{P})$. The solutions are defined to agree iff each active variable in the solution to $\mathcal{P}$ is assigned the same value in the solution to $\mathrm{T}(\mathcal{P})$ as in the solution to $\mathcal{P}$, and all other variables of $\mathcal{P}$ are assigned the value NULL. A simple update consists of an arbitrary number of either i) additions of constraints, removals of allowed tuples from constraints, removals of values from domains, and removals of variables with their domains, or ii) removals of constraints, additions of allowed tuples to constraints, additions of values to domains and additions of variables with their domains. In other words, a simple update cannot combine changes of the first and second type. In the case of removing variables, the constraints and assignments are projected to the remaining variables appropriately.

Theorem 1. *DCSP is not modularly representable by CSP.*

Proof. (Sketch) Consider the DCSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ with $\mathcal{V} = \{v_1, v_2\}$, $\mathcal{V}_I = \{v_1\}, D_1 = \{a\}, D_2 = \{b\}, \mathcal{C}_C = \mathcal{C}_A = \emptyset$, and assume it can be modularly represented by a CSP. Hence, there is a CSP $\mathrm{T}(\mathcal{P})$ such that in all the solutions of $\mathrm{T}(\mathcal{P})$ $v_1 = a$ and $v_2 = NULL$, as that is the only solution to $\mathcal{P}$. Consider now adding v_2 to $\mathcal{V}_I$. The resulting DCSP has only one solution where $v_1 = a$ and $v_2 = b$. This means that $\mathrm{T}(\mathcal{P})$ updated with v_2 must not have a solution in which $v_1 = a$ and $v_2 = NULL$. In addition, $\mathrm{T}(\mathcal{P})$ updated with v_2 must have at least one solution in which $v_1 = a$ and $v_2 = b$. It can be shown that simple updates cannot both add solutions and remove them, which is a contradiction and hence the assumption is false.

This result is caused by the condition that activity constraints or initial variables have to justify the activity of a variable in a solution. We note that there is a modular mapping from a CSP to DCSP, since CSP is a special case of DCSP. Thus, DCSP is strictly more expressive than CSP.

4 Generalized Definition of DCSP

In this section we give a new, generalized definition of DCSP. The activity constraints of the original DCSP are extended with disjunctive activity constraints and default negation. It is shown that a straightforward extension of the original definition by, e.g., disjunctive activity constraints would lead to significantly higher complexity. The new definition of a solution to DCSP utilizes a fixpoint condition instead of the minimality condition on the solutions. This allows a straightforward analysis and generalization of DCSP.

As noted in the introduction, the activity constraints of DCSP are not particularly expressive. For instance, in some configuration tasks a functional requirement can be satisfied by any of a given set of components, which would require disjunctive activity constraints [5]. Another case that cannot be represented in a straightforward manner is that a variable is active under the condition that some variables are not active or not given particular values in an assignment, i.e., under a condition referring to the complement of activity or constraints. The complement of activity can also be used to represent expressive requirements on the activity of variables in the desired solutions. Both these extensions can be included without increasing computational complexity (see Section 5).

We would thus like to extend DCSP to allow *generalized activity constraints* of the following form:

$$c_1, \ldots, c_j, not(c_{j+1}), \ldots, not(c_k) \overset{ACT}{\rightarrow} m\{v_1 \mid \ldots \mid v_l\}n \tag{1}$$

where $0 \leq m \leq n$. Intuitively, this activity constraint states that if constraints $c_1, \ldots, c_j$ are active in and satisfied by an assignment and the constraints $c_{j+1}, \ldots, c_k$ are not satisfied or not active in the assignment, then for the subset of the variables $v_1, \ldots, v_l$ active in the assignment, the cardinality of the subset is between m and n. If $m = 1$ and $n = l$, this becomes an inclusive disjunction of the variables. On the other hand, if $m = n = 1$, the right hand side becomes an exclusive disjunction. We also allow rules with $m = 0$ for representing *optional* variables, i.e., variables that may be active or inactive. Always require activity constraints are again treated as short hand notation.

The original definition does not seem to provide a promising basis for generalizing DCSP with such activity constraints. It employs a minimality condition that makes it hard to extend basic DCSP without a substantial increase in computational complexity. For example, disjunctive always requires activity constraints of the form

$$v_1, \ldots, v_m \stackrel{ACT}{\rightarrow} 1\{v_{m+1} \mid \ldots \mid v_{m+n}\}n \tag{2}$$

already lead to $\Sigma_2\mathbf{P}$-hardness of $DCSP_R(D)$.

Theorem 2. *$DCSP_R(D)$ is $\Sigma_2\boldsymbol{P}$-hard for disjunctive activity constraints of the form (2). This holds even in the case where all variables have unary domains, there are no other constraints, and the requirements consist of only one compatibility constraint.*

Proof. (Sketch) This result can be established by a reduction from the problem of deciding whether a positive disjunctive database has a minimal model containing a given atomic formula, known to be $\Sigma_2\mathbf{P}$-complete [1]. The reduction is based on the following observation. If we map every rule $a_1 \vee \cdots \vee a_m \leftarrow a_{m+1}, \ldots, a_n$ of a database to an activity constraint $a_{m+1}, \ldots, a_n \stackrel{ACT}{\rightarrow} 1\{a_1 \mid \ldots \mid a_m\}m$ where each a_i is taken as a DCSP variable with a unary domain, then minimal models of the database coincide with minimal solutions of the corresponding DCSP.

In order to extend the basic DCSP formalism without increasing the complexity we employ a definition based on a fixpoint equation of an operator on the lattice formed by the set of assignments and their subset relation. However, we first define when an assignment satisfies a generalized activity constraint. The definition of when an assignment satisfies compatibility constraints remains the same as previously.

Definition 5. *A legal assignment $\mathcal{A}$ for a DCSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ satisfies the activity constraints in $\mathcal{P}$ iff for each activity constraint in $\mathcal{C}_A$ of the form (1) the following holds: if $c_1 \ldots c_j$ are active and satisfied, and $c_{j+1}, \ldots, c_k$ are not satisfied by or not active in $\mathcal{A}$, then for the set $V \subseteq \{v_1, \ldots, v_l\}$ active in $\mathcal{A}$, $m \leq\mid V \mid\leq n$ holds.*

The fixpoint condition in the new definition is another way of ensuring that active variables are justified by initial variables and activity constraints. To capture this, we first define a reduction of a set of generalized activity constraints and the initial variables with respect to an assignment. The intuition behind the reduction is that, given an activity constraint, if a variable on the right hand side of it is active in a solution and the negated activity constraints are either not satisfied or not active, then the reduct includes a simpler *instantiated activity constraint* that *can* justify the activity of the variable. If a variable in the head of an activity constraint is not active, then there is no need for its activity to be justified and consequently no corresponding instantiated activity constraints are included.

More precisely, the reduct of each activity constraint contains one instantiated activity constraint for every active variable that occurs on its right hand side, if all the constraints with default negation on the left hand side are either not satisfied or not active in the assignment. The left hand side of the instantiated activity constraint is otherwise the same as that of the activity constraint, except that the constraints with default negation are removed. This treatment of default negation is similar to the stable model semantics of logic programs [2]. Further, the variables on the right hand side of the instantiated activity constraints are replaced by their assignments. Initial variables are also given a special activity constraint form, the left hand side of which is satisfied by every assignment.

Definition 6. *We define the reduct $(\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}}$ of a set of generalized activity constraints $\mathcal{C}_A$ and a set of initial variables $\mathcal{V}_I$ w.r.t. an assignment $\mathcal{A}$ as follows:*

$$\begin{aligned}(\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}} = \{ & c_1, \ldots, c_i \overset{ACT}{\rightarrow} v_j = d_{jp} \mid \\ & (c_1, \ldots, c_i, not(c_{i+1}), \ldots, not(c_k) \overset{ACT}{\rightarrow} m\{v_1 \mid \ldots \mid v_l\}n) \in \mathcal{C}_A, \\ & (v_j = d_{jp}) \in \mathcal{A} \textit{ for some } j \textit{ where } 1 \leq j \leq l, \\ & c_{i+1}, \ldots, c_k \textit{ are not satisfied by or active in } \mathcal{A}\} \\ & \cup \{ \overset{ACT}{\rightarrow} v_j = d_{jk} \mid v_j \in \mathcal{V}_I, (v_j = d_{jk}) \in \mathcal{A}\} .\end{aligned}$$

We denote the set of instantiated activity constraints resulting from the reduct $(\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}}$ by the shorthand notation $\mathcal{C}_I$ for brevity. We now define an operator on the lattice formed by the set of all possible assignments and the subset relation on these. The operator intuitively captures how the instantiated activity constraints introduce new active variables to an assignment when their left hand side constraints are active and satisfied.

Definition 7. *Given a set of instantiated activity constraints $\mathcal{C}_I$, the operator $T_{\mathcal{C}_I}(\cdot)$ on an assignment $\mathcal{A}$ is defined as follows:*

$$\begin{aligned}T_{\mathcal{C}_I}(\mathcal{A}) = \{v = d \mid \ & (c_1, \ldots, c_k \overset{ACT}{\rightarrow} v = d) \in \mathcal{C}_I, \\ & c_i \textit{ is active in and satisfied by } \mathcal{A}, 1 \leq i \leq k\}\end{aligned}$$

Intuitively, a solution is a legal assignment that satisfies the constraints in the DCSP and contains all and only the active variables that are justified by the set of initial variables and activity constraints. Since active variables may be justified recursively, a solution to a DCSP is defined as a fixpoint of the above operator for a given reduct. A *fixpoint* q of an operator $\tau(\cdot)$ is such that $\tau(q) = q$. This ensures that every variable with a justification is active in the solution.

The operator $T_{\mathcal{C}_I}$ is monotonic, i.e., for assignments $\mathcal{A}_i$ and $\mathcal{A}_j$, $\mathcal{A}_i \subseteq \mathcal{A}_j$ implies $T_{\mathcal{C}_I}(\mathcal{A}_i) \subseteq T_{\mathcal{C}_I}(\mathcal{A}_j)$. This can be seen by noting that if the left hand side of an instantiated activity constraint is satisfied by an assignment, it is satisfied by all its supersets as well. A monotonic operator has a unique *least fixpoint* that can be computed by iteratively applying the operator, starting from the empty set [4]. As a solution must not contain unjustified active variables, it should be such a least fixpoint, denoted by $\mathrm{lfp}(T_{\mathcal{C}_I})$. This ensures that all the assignments in a solution are justified by the instantiated activity constraints.

Definition 8. *Given a DCSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_I, \mathcal{C}_C, \mathcal{C}_A \rangle$ a legal assignment $\mathcal{A}$ is a solution to $\mathcal{P}$ iff i) $\mathcal{A}$ satisfies the constraints in $\mathcal{P}$, ii) the initial variables are active in $\mathcal{A}$, iii) $\mathcal{A} = \mathrm{lfp}(T_{(\mathcal{C}_A,\mathcal{V}_I)^{\mathcal{A}}})$.*

Example 3. Consider again the mixer DCSP of Example 1. The assignment $\mathcal{A}_2 = \{Mi = r, Mt = b, Vol = s, Coo = cool\}$ is still a solution according to the Definition 8 since it satisfies the constraints, the initial variables are active in it, and it is the least fixpoint of the operator. The last property can be established as follows. First, the reduct of the activity constraints is computed with respect to $\mathcal{A}_2$:

$$\begin{aligned}&\mathcal{C}_I = (\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}_2} = \\ &\{(\stackrel{ACT}{\rightarrow} Mi = r), (\stackrel{ACT}{\rightarrow} Mt = b), (\stackrel{ACT}{\rightarrow} Vol = s), (Mi = r \stackrel{ACT}{\rightarrow} Coo = cool)\} \ .\end{aligned}$$

Then, the operator is applied iteratively starting from the empty set:

$$\begin{aligned}T_{\mathcal{C}_I}(\emptyset) &= \{Mi = r, Mt = b, Vol = s\} = \mathcal{A}_{21}\\ T_{\mathcal{C}_I}(\mathcal{A}_{21}) &= T_{\mathcal{C}_I}(T_{\mathcal{C}_I}(\emptyset)) = \{Mi = r, Mt = b, Vol = s, Coo = cool\} = \mathcal{A}_{22}\\ T_{\mathcal{C}_I}(\mathcal{A}_{22}) &= T_{\mathcal{C}_I}(T_{\mathcal{C}_I}(T_{\mathcal{C}_I}(\emptyset))) = \{Mi = r, Mt = b, Vol = s, Coo = cool\}\end{aligned}$$

Thus, $\mathcal{A}_{22}$ is the least fixpoint and $\mathrm{lfp}(T_{\mathcal{C}_I}) = \mathcal{A}_2$.

On the other hand, the assignment $\mathcal{A}_3 = \{Mi = r, Mt = b, Vol = s, Coo = cool, Con = con2\}$ is not a solution. It does satisfy the constraints, but $\mathcal{C}_I = (\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}_3} = \{(\stackrel{ACT}{\rightarrow} Mi = r), (\stackrel{ACT}{\rightarrow} Mt = b), (\stackrel{ACT}{\rightarrow} Vol = l), (Mi = r \stackrel{ACT}{\rightarrow} Coo = cool), (Mt = d \stackrel{ACT}{\rightarrow} Con = con2)\}$. The least fixpoint can again be constructed similarly as above, and $\mathrm{lfp}(T_{\mathcal{C}_I}) = \{Mi = r, Mt = b, Vol = s, Coo = cool\} \neq \mathcal{A}_3$.

Example 4. In Example 1, the configuration $\{Mi = r, Mt = d, Vol = s, Coo = cool, Con = con2\}$ with a condenser and a cooler is a legal configuration. In order to avoid having both condenser and cooler active, we consider the following DCSP obtained by replacing the activity constraint a_1 by an "exclusively disjunctive" activity constraint a_3:

$$\begin{aligned}a_2 &= (Mt = d \stackrel{ACT}{\rightarrow} Con)\\ a_3 &= ((c_4 \stackrel{ACT}{\rightarrow} 1\{Coo \mid Con\}1)\\ c_1(Con, Vol) &= \{(con1, l), (con2, l), (con2, s)\}\\ c_2(Mi, Vol) &= \{(r, s), (m, s), (m, l), (t, s), (t, l)\}\end{aligned}$$

where $c_4(Mi, Mt) = \{(r, d)\}$. An assignment $\mathcal{A}_1 = \{Mi = r, Mt = d, Vol = s, Con = con2\}$ is a solution since it clearly satisfies the constraints, the initial variables are active in it, and for the reduct $(\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}_1} = \{(\stackrel{ACT}{\rightarrow} Mi = r), (\stackrel{ACT}{\rightarrow} Mt = d), (\stackrel{ACT}{\rightarrow} Vol = s), (Mt = d \stackrel{ACT}{\rightarrow} Con = con2), (c_4 \stackrel{ACT}{\rightarrow} Con = con2)\}$, $\mathrm{lfp}(T_{(\mathcal{C}_A,\mathcal{V}_I)^{\mathcal{A}_1}}) = \mathcal{A}_1$. On the other hand, $\mathcal{A}_2 = \{Mi = r, Mt = d, Vol = s, Con = con2, Coo = cool\}$ is not a solution since it does not satisfy the activity constraint a_3 that only one of the variables Coo and Con should be active. If the activity constraint $a_4 = (not(Coo = cool) \stackrel{ACT}{\rightarrow} 1\{Con\}1)$ with default negation were further added to the example, the reduct with respect to $\mathcal{A}_1$ would be as follows: $\{(\stackrel{ACT}{\rightarrow} Mi = r), (\stackrel{ACT}{\rightarrow} Mt = d), (\stackrel{ACT}{\rightarrow} Vol = s), (Mt = d \stackrel{ACT}{\rightarrow} Con = con2), (c_4 \stackrel{ACT}{\rightarrow} Con = con2), (\stackrel{ACT}{\rightarrow} Con = con2)\}$. The last instantiated activity constraint is the reduct of a_4. Thus the assignment $\mathcal{A}_1$ would still remain a solution. This latter type of activity constraint can be seen as a default rule stating that, all things being equal, one should choose a condenser over cooler, if possible.

Example 5. Consider now the DCSP which is obtained by replacing the activity constraint a_1 in Example 1 by an "inclusively disjunctive" activity constraint $a_3' = (c_4 \stackrel{ACT}{\rightarrow} 1\{Coo \mid Con\}2)$ with c_4 a constraint $\{(r, d)\}$ defined on Mi and Mt. Now, the assignment $\mathcal{A}_2 = \{Mi = r, Mt = d, Vol = s, Con = con2, Coo = cool\}$ is a solution, since it satisfies the constraints, the initial variables are active in it, and the reduct $(\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}_2} = \{(\stackrel{ACT}{\rightarrow} Mi = r), (\stackrel{ACT}{\rightarrow} Mt = d), (\stackrel{ACT}{\rightarrow} Vol = s), (Mt = d \stackrel{ACT}{\rightarrow} Con = con2), (c_4 \stackrel{ACT}{\rightarrow} Con = con2), (c_4 \stackrel{ACT}{\rightarrow} Coo = cool)\ \}$. The last two instantiated activity constraints are the reduct of a_3', since both cooler and condenser are active in the assignment. Now $\mathrm{lfp}(T_{(\mathcal{C}_A, \mathcal{V}_I)^{\mathcal{A}_2}}) = \mathcal{A}_2$. Also the assignment $\mathcal{A}_1$ in the previous example remains a solution.

In our definition, we in fact relax the minimality requirement. Each active variable in a solution is justified by the activity constraints or initial variables, but a disjunctive activity constraint may justify more variables than a subset minimal solution would contain, as demonstrated by Example 5.

5 Computational Complexity of DCSP

In this section we analyze the complexity of the relevant decision problems for our generalization of DCSP. This also provides a complexity result for the original definition. We first show that our generalization is equivalent to the original definition for the activity constraints handled by it.

Theorem 3. *Definition 8 of a solution to a DCSP is equivalent to Definition 3 for problems without disjunctive activity constraints and default negation, and with only one constraint on the left hand side of every activity constraint.*

Proof. (Sketch) The difference between the two definitions is that the minimality condition of the original definition is replaced by the fixpoint condition in Definition 8. In order to prove the equivalence it remains to show that the fixpoint condition is equivalent to subset minimality when original activity constraints are allowed. This can be done using the following observations. A solution according to Definition 3 is a minimal set satisfying the constraints and making the initial variables active. Hence, it is contained in a solution according to Definition 8 which also satisfies the constraints and makes the initial variables active.

On the other hand, a solution according to Definition 8 is contained in a solution according to Definition 3 as it can be constructed iteratively by starting from the empty solution and by applying the operator T. It can be shown inductively that the result of this iteration is contained in a solution according to Definition 3.

Using the fixpoint definition of a solution it is straightforward to establish that both $DCSP(D)$ and $DCSP_R(D)$ remain **NP**-complete for the generalized DCSP. This is in contrast to the complexity of $DCSP_R(D)$ for the straightforward extension sketched in the beginning of Section 4.

Theorem 4. *$DCSP(D)$ and $DCSP_R(D)$ are **NP**-complete for the generalized DCSP.*

Proof. (Sketch) First, we note that $DCSP(D)$ is a special case of $DCSP_R(D)$ with an empty set of requirements. Then, we argue that both $DCSP(D)$ and $DCSP_R(D)$ are **NP**-hard since CSP, which is known to be **NP**-complete, is a special case of DCSP with $\mathcal{V}_I = \mathcal{V}$ and no activity constraints. Further, $DCSP_R(D)$ (and thus also $DCSP(D)$) is **NP**-complete because it is in **NP**. The containment in **NP** is due to the fact that whether an assignment is a solution and satisfies a set of requirements can be checked in polynomial time. This result can be shown by noting that whether an assignment satisfies a set of activity and compatibility constraints and whether it is the least fixpoint of the operator on the instantiated activity constraints can be both decided in polynomial time. The latter property holds since the reduct can obviously be computed in polynomial time, by processing one rule at a time, and the least fixpoint of the operator can be computed in polynomial time.

By Theorem 3 the original DCSP is a special case of the generalized DCSP and we obtain similarly the following corollary.

Corollary 1. *$DCSP(D)$ and $DCSP_R(D)$ are **NP**-complete for the original DCSP, i.e. without disjunctive activity constraints and default negation.*

6 Implementation

In this section, we briefly discuss two novel solution methods for the original DCSP. The first is similar to the original algorithm described in [5] whereas the second is based on mapping a DCSP to a type of propositional logic programs [10]. To test the performance of both algorithms, we use a set of examples from the configuration domain, CAR [5], CARx2 [10], a simplified form of a hospital monitor problem [11] and a simplified form of the mixer problem [14]. These problems are characterized in Table 2 by the number of variables, number of compatibility constraints and their maximum arity, number of activity constraints and the maximum arity of their left hand side constraints, maximum domain size, number of solutions, and size of the initial search space calculated by multiplying the domain sizes of the variables.

Table 2. Characteristics of the examples.

Problem	$\mid \mathcal{V} \mid$	$\mid \mathcal{C}_C \mid$ / max. arity	$\mid \mathcal{C}_A \mid$ / max. arity	$max(\mid \mathcal{D}_i \mid)$	$\mid solutions \mid$	search space
CAR	8	7 / 3	8 / 1	3	198	1296
CARx2	8	7 / 3	8 / 1	6	44456	331776
Monitor	24	9 / 3	19 / 3	4	1320	196608
Mixer	8	4 / 2	6 / 1	4	88	1152

The first implementation differs from the one described in [5] in that it does not use an ATMS. The algorithm is based on a simple backtracking algorithm used to solve standard CSPs. As long as all variables are not yet assigned a value the algorithm chooses the next variable and a value to assign to that variable such that the value is still consistent with the compatibility constraints. Then the algorithm checks if some activity constraint has become relevant, i.e., if the left hand side of an activity constraint is activated and satisfied by the already assigned values. In that case, new variables are activated and added to the list of not yet assigned variables. If all values of a variable have been considered, the algorithm backtracks to the last variable that still has values left. Variables that have been activated by an activity constraint based on the value of a variable deassigned in the backtracking step have to be deactivated as well. The algorithm continues until no more activity constraints can be activated and all currently active variables have been assigned a value. In the current implementation, no consistency algorithms are applied. The implementation was written in Java (Sun's JDK) and run on a Pentium II, 266 MHz, with 96MB of memory, Windows NT operating system. The implementation and the test problems are available at `www.cs.hut.fi/~pdmg/CP99/`.

The second implementation is based on first translating the DCSP to a set of rules in Configuration Rule Language, a type of propositional logic programs [10], for which the solutions are found using an efficient C++ implementation of the stable model semantics of normal logic programs [6]. The translation results in a set of rules whose size is linear in the size of the DCSP, in fact roughly the same. Solutions are found using backtracking search through a binary search tree where nodes are Boolean valued variables modelling assignments of values to DCSP variables. The search space is extensively pruned using rule propagation. The tests were run on a Pentium II 266 MHz with 128MB of memory, Linux 2.2.3 operating system, `smodels` version 2.22 and `lparse` version 0.99.22. Further details can be found in [10], the implementation at `www.tcs.hut.fi/pub/smodels` and the test problems at `www.tcs.hut.fi/pub/smodels/tests/cp99.tar.gz`.

The time to find the first and all solutions are given in Table 3. We note that for the first implementation, the problem inputs and outputs are handled in main memory, whereas the results for the second implementation include the time for reading the input from a file and parsing it. However, the execution time for the second implementation does not include the time for translating the DCSP to **CRL** form, which was done manually. Since it is fairly difficult to compare the execution time of algorithms written in different languages, executed on different operating systems and based on different problem formulations, we provide as an additional characterization the number of non-deterministic guesses G for finding the first, respectively all, solutions in Table 3. In a non-deterministic guess of the backtracking algorithm, one variable is assigned a value and it is checked if the resulting (partial) assignment (together with the set of already assigned variables) remains consistent. In the **CRL**-based implementation, a non-deterministic guess chooses a Boolean valued variable corresponding to a particular value assignment of a DCSP variable.

Table 3. Results for the examples.

Implementation	first solution	all solutions	G^{first}	G^{all}
CAR backtrack-based	0.04 s	0.07 s	47	624
CAR **CRL**-based	0.05 s	0.06 s	5	197
CARx2 backtrack-based	0.01 s	11.8 s	217	71230
CARx2 **CRL**-based	0.05 s	3.2 s	6	44455
Monitor backtrack-based	< 1 ms	0.08 s	21	15060
Monitor **CRL**-based	0.06 s	0.31 s	9	1319
Mixer backtrack-based	< 1 ms	0.01 s	15	484
Mixer **CRL**-based	0.04 s	0.05 s	4	87

7 Conclusions and Future Work

Dynamic constraint satisfaction problems were introduced to capture dynamic aspects in generating problem spaces to combinatorial problems such as product configuration [5]. We argue that DCSP is indeed a more expressive representation than CSP for such problems. However, the activity constraints of DCSP are relatively restricted. We present a generalized definition of DCSP which allows disjunctions and default negation in activity constraints. We then show that the relevant decision problems remain **NP**-complete for the generalization, which employs a fixpoint condition instead of a minimality condition used in the original definition. Thus DCSP seems to be a more feasible framework for dynamic problems than CSP. We also show that a straightforward generalization of the original definition would increase the complexity significantly. As further work, the expressivity and computational complexity of combining the different elements of our generalization, minimality and further forms of optimization criteria should be analyzed.

There are few reports on implementations of algorithms for the original DCSP. We sketch two implementations, one based on a modified backtracking algorithm for CSP, the other based on mapping DCSP to a type of propositional logic program rules. The test results for both on a set of simple configuration problems are acceptable and indicate that they should be further pursued. Both implementations should be extended with the generalizations and empirically tested on larger problems. The logic program based implementation employs rule propagation at each point in search space which seems to reduce considerably the number of non-deterministic guesses needed for finding solutions. Hence, the backtrack implementation of DCSP can probably be enhanced similarly by integrating to it various consistency algorithms similar to standard CSP search algorithms.

Acknowledgements. The work of the first author has been supported by the Helsinki Graduate School in Computer Science and Engineering and the Technology Development Centre Finland, and that of the third author by the Academy of Finland (Project 43963). The work of the second author is a continuation of that done at the AI-Lab, EPFL. She would like to thank its director, Professor B. Faltings, and the colleagues there who contributed to the work.

References

1. T. Eiter and G. Gottlob. Complexity aspects of various semantics for disjunctive databases. In *Proc. of the 12th Symposium on Principles of Database Systems*, pages 158–167, 1993.
2. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th Intern. Conf. on Logic Programming*, pages 1070–1080, 1988.
3. E. M. Gelle. *On the generation of locally consistent solution spaces.* Ph.D. Thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1998.
4. J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.
5. S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. of the 8th National Conf. on Artificial Intelligence*, pages 25–32, 1990.
6. I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. of the 4th International Conf. on Logic Programming and Non-Monotonic Reasoning*, pages 420–429, 1997.
7. D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. In *Configuration – Papers from the 1996 Fall Symposium. AAAI Technical Report FS-96-03*, 1996.
8. D. Sabin and R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems & Their Applications*, pages 42–49, July/August 1998.
9. T. Soininen and E. M. Gelle. Dynamic constraint satisfaction in configuration. In *AAAI'99 Workshop on Configuration. To appear as AAAI Technical Report*, 1999.
10. T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Practical Aspects of Declarative Languages (PADL99), LNCS 1551.* Springer-Verlag, 1999.
11. T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen. Towards a general ontology of configuration. *AI EDAM*, 12:357–372, 1998.
12. M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12:307–320, 1998.
13. E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, 1993.
14. M. van Velzen. A Piece of CAKE, Computer Aided Knowledge Engineering on KADSified Configuration Tasks. Master's thesis, University of Amsterdam, Social Science Informatics, 1993.