Dynamic Constraint Satisfaction Problems

Sanjay Mittal and **Brian Falkenhainer**
System Sciences Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto CA 94304

Abstract

Constraint satisfaction (CSP) is a powerful and extensively used framework for describing search problems. A CSP is typically defined as the problem of finding consistent assignment of values to a fixed set of variables given some constraints over these variables. However, for many synthesis tasks such as configuration and model composition, the set of variables that are relevant to a solution and must be assigned values changes dynamically in response to decisions made during the course of problem solving. In this paper, we formalize this notion as a *dynamic constraint satisfaction problem* that uses two types of constraints. *Compatibility constraints* correspond to those traditionally found in CSPs, namely, constraints over the values of variables. *Activity constraints* describe conditions under which a variable may or may not be actively considered as a part of a final solution. We present a language for expressing four types of activity constraints in terms of variable values and variables being considered. We then describe an implemented algorithm that enables tight interaction between constraints about variable activity and constraints about variable values. The utility of this approach is demonstrated for configuration and model composition tasks.

1 Introduction

Constraint satisfaction is a powerful and extensively used framework for expressing and solving search problems. A variety of general techniques have been developed for finding a consistent assignment of values to a predefined set of variables [3; 8; 9; 12; 11]. The variables typically have preenumerated domains of discrete values, and a set of constraints over subsets of these variables limit their possible values.

In contrast, for synthesis tasks such as configuration, design, or model composition, the constraint problem is of a more dynamic nature where any of the elements of the constraint problem (i.e., variables, domains, constraints) might change as the search progresses. For example, selecting the type of hard disk controller for a computer configuration is only relevant when a hard disk has been chosen as the form of secondary storage. Installing a floppy disk drive would require solving for a different set of variables and constraints.

A common approach in such cases is to embed general constraint satisfaction mechanisms within a larger task-specific problem solving architecture. In this coupled mode of problem-solving, the problem solver makes decisions about which variables and constraints are relevant in order to define a problem solvable by some constraint satisfaction mechanism. For example, in the Cossack expert system for configuration [6],the problem solver created variables in response to requirements specifying some functional aspects of a desired configuration. Consistent choices were assigned to these variables by selecting a component and propagating constraints associated with the selected component. Separate mechanisms were used for creating variables and processing constraints on them. Similarly, the Mapsee-2 and Mapsee-3 programs for understanding sketch maps [16] have a control cycle that alternates between creating a new variable for each image element (a chain of edges in the Mapsee terminology) and propagating constraints between existing variables to find consistent interpretations for the image elements. However, experience with Cossack and Mapsee-2 suggests that this separation becomes cumbersome and inefficient when decisions about which variables are relevant interact closely with decisions about consistent value assignments.

In this paper, we identify a specialized but useful class of dynamic problems that we call *dynamic constraint satisfaction problems* (DCSP). For this class of problems, we propose a more integrated approach that extends the notion of constraint satisfaction problems to include constraints about the variables considered in each solution. When a variable must be included, we say that it is *active*. By expressing the conditions under which variables are and are not active, standard CSP methods can be extended to make inferences about variable activity as well as their possible value assignments.

The key characteristic of such dynamic constraint problems is that constraints on introducing or removing variables from a potential solution closely interact with constraints on consistent assignment of values to some already identified set of variables. This property distinguishes our model of DCSP from other "dynamic" formalisms such as incremental constraint satisfaction methods that can handle a changing set of variables [17] or constraints [7], but cannot themselves reason about the activity of a variable. Similarly, constraint logic programming [10] naturally allows variables to be dynamically introduced but provides no special support for efficiently reasoning about variable activity.

We begin by considering how to formalize configu-

ration as a constraint satisfaction problem. This provides the motivation for the formal definition of dynamic CSPs. We describe a simple language for stating constraints about the activity of problem variables in a form that is usable within an extended constraint satisfaction framework. This enables introduction and removal of variables and constraints during search. Further, it provides a useful framework in which to express certain kinds of knowledge for synthesis problems. Examples from configuration and model composition tasks are used to illustrate the use of our language. Finally, we describe an implemented search algorithm for solving DCSPs that is tuned to this specialized language. We close by describing some extensions in progress.

2 Modeling configuration as constraint satisfaction

A constraint satisfaction problem (CSP) is typically specified by a set of variables $V = \{v_1,\cdots,v_n\}$ and a set of constraints C on subsets of V limiting the values that may be assigned in a consistent manner [11; 12]. Each variable v_i has an associated domain $D_i = \{d_{i1},\cdots,d_{in_i}\}$ which identifies its set of possible values. The constraint satisfaction task is to find assignments of values for $\{v_1,\cdots,v_n\}$ that simultaneously satisfy all the constraints C.

We shall motivate the ideas behind this paper by considering how to formalize a configuration task as defined in [14] as a CSP. In that paper, it was shown that many configuration domains satisfy two important properties. One, the functional roles played by a component in a configured artifact are known ahead of time. Two, for each functional role, one can identify a set of components as the "key" for each role, i.e., one of these "key components" are always needed for implementing the corresponding functional role. For example, in a computer domain, the functional roles include instruction execution, program memory, secondary storage, display, and printing. The key components for these functions respectively include microprocessor chip, memory chip, hard disk or floppy disk, display terminal, and printer.

One can model such configuration tasks as a CSP by making additional simplifying assumptions such as: (1) there are a fixed set of functional roles to be decided in a configured artifact; (2) there is a one-to-one association between functional roles and the components that function in that role; and (3) there is a fixed set of components for each functional role. In the CSP model, we create a variable for each functional role and the components associated with that functional role constitute the domain of these variables. Constraints are used to represent compatibility and selection knowledge.

The model presented above is rather simplistic, since only assumption (3) above is valid. Assumption (2) is generally invalid, since the mapping between functional roles and actual available components is typically many-to-many. For example, in order to provide secondary storage one not only needs a disk drive but also a drive controller, bus connection, and some associated driver software. Here the crucial problem is not just the additional components that are needed but the fact that *different components for the same functional role often need non-identical sets of additional components.* Similarly, some of the available components often provide more than one function. For example, the motherboard on a PC often implements many functions such as the microprocessor, co-processor, program memory, basic I/O system, and assorted other functions. The problem is not just the multiplicity of functions but the fact that such *multi-function components often provide non-identical sets of functions.*

One can use the "key component" idea to partially solve this problem. Essentially, retain the mapping described above but keep in mind that the mapping is only partial, i.e., the component associated with a functional role may only partially provide that function and other components (or functional roles) may be needed to complete the requirements. One can express these additional requirements by dynamically introducing "new" variables and constraints in the solution as a result of selecting a partially satisfying component C for a functional role R. The model of dynamic CSP presented in this paper formalizes this insight by extending CSP to allow constraints on both the values of a variable and "relevance" to a solution. It is important to point out that the language presented here is more general than the initial motivation. In particular, as will become apparent from the examples discussed later in the paper, the same constraint type can be used to represent many different kinds of domain knowledge. Furthermore, the CSP extensions require new methods for solving the problem and also create opportunities for additional heuristics for controlling search.

3 Dynamic constraint satisfaction problems

In CSPs, the sets V, D_i, and C are fixed and known beforehand. Each solution must contain an assignment for every variable in V. In DCSPs, there is also some set of variables V. However, unlike for a CSP, not all variables have to be assigned a value to solve the problem. Some variables and their value assignments render other variables irrelevant, thus indicating that effort should not be spent considering values and constraints for these irrelevant variables.

Because the set of variables requiring assignment is not fixed by the problem definition, we distinguish between variables that appear in the solution and those that do not. A variable is called *active* when it must be part of the solution. The statement that v_i is active is represented by the proposition $active{:}v_i$; the statement that v_i is not active is represented by $\neg active{:}v_i$. A variable v_i is assigned one of its possible values $\{d_{i1},\cdots,d_{in_i}\}$ if and only if it is active:

$$active{:}v_i \leftrightarrow v_i = d_{i1} \vee \cdots \vee v_i = d_{in_i}$$

DCSPs can themselves post constraints about which variables are active. Thus, a DCSP explicitly constructs each *active*:v_i and every solution to a DCSP only assigns values to variables that are also active.

A dynamic CSP specifies a non-empty set of *initial variables* V_I that must appear in all solutions (i.e., for all v_i in V_I, *active*:v_i always holds). The set of active variables appearing in each solution will always be a superset of V_I. A problem not requiring additional variables reduces to a conventional CSP with $V = V_I$.

To concisely represent the conditions under which a variable and its associated domain become relevant to forming a solution, we extend the notion of constraint to include a set of *activity constraints* C^A affecting variable activity. These constrain a variable to be active or not active based on other variables' activity and value assignments. There are several activity constraint types described in section 4. The simplest of these is logically equivalent to:

$$P(v_1,\cdots,v_n) \rightarrow active{:}v_j$$

where $v_j \notin \{v_1,\cdots,v_n\}$ and P is a predicate over variables and their possible values.

To distinguish between activity constraints and traditional constraints over variable values, we call the standard set of CSP constraints *compatibility constraints*, represented by C^C. Due to the active variable distinction, we must also revise the conditions under which a compatibility constraint is satisfied. Take $P(v_i,v_j,v_k)$ to represent a compatibility constraint over variables v_i, v_j, and v_k. It is treated as being logically equivalent to:

$$active{:}v_i \wedge active{:}v_j \wedge active{:}v_k \rightarrow P(v_i,v_j,v_k)$$

Thus, if one of the variables in a constraint is not active, the constraint is trivially satisfied; if all of the variables in a constraint are active, it has the standard meaning and we say that the constraint is *active*. For efficiency reasons, we make these two cases explicit within the problem solver so that only active constraints need be checked.

We can now define a dynamic constraint satisfaction problem as follows:

Given:

- A set of variables V representing all variables that may potentially become active and appear in a solution. (This list need not be explicitly preenumerated, although our algorithm does not currently handle that case.)
- A non-empty set of *initial variables* $V_I = \{v_1,\cdots,v_k\}$, which is a subset of V.
- A set of discrete, finite domains $D_1,\cdots,D_k$, with each domain $D_i = \{d_{i1},\cdots,d_{ik_i}\}$ representing the set of possible values for each variable v_i in V.
- A set of *compatibility constraints* C^C on subsets of V limiting the values they may take on.
- A set of *activity constraints* C^A on subsets of V specifying constraints between the activity and possible values of problem variables.

Find:

- All solutions, where a solution is an assignment $\mathcal{A}$ which meets two criteria:
  1. The variables and assignments in $\mathcal{A}$ satisfy $C^C \cup C^A$.
  2. No subset of $\mathcal{A}$ is a solution.

This definition extends the standard definition of CSP by introducing activity constraints and identifying a subset of the possible variables as initial variables. Note that the choice of initial variables is important for defining the problem. Keeping everything else the same, we can create different problems simply by changing V_I.[1] Also note that because activity may be affected by value assignments, each possible solution may contain a different set of variables.

Consider the following simple dynamic CSP. We are given the variables v_1, v_2, v_3, v_4 with domains $D_1 = \{a,b\}$, $D_2 = \{c,d\}$, $D_3 = \{e,f\}$, and $D_4 = \{g,h\}$. Further, we are given initial variables $V_I = \{v_1,v_2\}$ and constraints:

$$v_1 = a \rightarrow v_2 = d$$
$$v_1 = b \rightarrow v_2 = c$$
$$v_2 = c \wedge v_3 = e \rightarrow v_4 = h$$
$$v_1 = b \rightarrow active : v_3$$
$$v_3 = e \rightarrow active : v_4$$

This problem has three solutions:

$$v_1 = a,\ v_2 = d$$
$$v_1 = b,\ v_2 = c,\ v_3 = f$$
$$v_1 = b,\ v_2 = c,\ v_3 = e,\ v_4 = h$$

4 Activity constraints

In applying the dynamic CSP framework to several application tasks, we have found it useful to define a specialized language for expressing four types of activity constraints. This language is useful for compactly representing selection and composition constraints in synthesis tasks. Additionally, separating these different activity constraints into distinct types also improves search efficiency by adding finer control over constraint propagation within our algorithm.

4.1 Require variable constraint

The most fundamental activity constraint is the *require variable* (RV), which establishes a variable's activity based on an assignment of values to a set of active variables. RV constraints have the form:

$$P(v_1,\cdots,v_j) \underset{\text{RV}}{\Longrightarrow} v_k \qquad (\text{where } v_k \notin \{v_1,\cdots,v_j\})$$

This notation is logically equivalent to:

$$P(v_1,\cdots,v_j) \rightarrow active{:}v_k$$

[1] This flexibility in reusing the knowledge base is often quite useful in the class of problems that have provided the motivation for the ideas presented here.

where $P(v_1,\cdots,v_j)$ is a predicate over the possible values of variables. When $P(v_1,\cdots,v_j)$ is both active (i.e., $v_1,\cdots,v_j$ are all active) and satisfied, v_k must be active. A contradiction results if it is inconsistent for v_k to be active.

One use of RV constraints is to represent conditional requirements in synthesis tasks.

4.2 Always Require

The *always require variable* (ARV) constraint extends the basic notion of a require constraint to require a variable's activity based on the activity of other variables, independent of their current value. ARV constraints have the form:

$$v_1 \wedge \cdots \wedge v_j \underset{\text{ARV}}{\Longrightarrow} v_k \qquad (\text{where } v_k \notin \{v_1,\cdots,v_j\})$$

This form handles the special case where all choices for v_1 through v_j require v_k. While logically equivalent to

$$\{v_1{=}d_{11} \vee \cdots \vee v_1{=}d_{1n_1}\} \wedge \cdots \wedge \{v_j{=}d_{j1} \vee \cdots \vee v_j{=}d_{jn_j}\} \rightarrow \mathit{active}{:}v_k$$

the special form is more compact and can lead to more efficient search. The search method presented in section 6 takes advantage of the form of ARV constraints to make v_k active as soon as v_1 through v_j become active, without waiting for particular value assignments to v_1 through v_j.

Note that the initial variables in the definition of a DCSP could alternatively be expressed as a set of ARV constraints whose conditions are "true". Thus, a given DCSP can be easily modified by simply changing such top-level ARV constraints.

4.3 Require Not

In addition to stating when variables must be in the problem space, some tasks require the ability to state when variables must not be in the problem space. For example, selecting a convertible body frame for a car is inconsistent with **any** type of sunroof. The *require not* (RN) constraint states an inconsistency between an assignment of values to a set of active variables and another variable being active. RN constraints have the form:

$$P(v_1,\cdots,v_n) \underset{\text{RN}}{\Longrightarrow} v_j \qquad (\text{where } v_j \notin \{v_1,\cdots,v_n\})$$

They are logically equivalent to

$$P(v_1,\cdots,v_n) \rightarrow \neg \mathit{active}{:}v_j$$

For some applications, this constraint is more efficiently reexpressed as an inconsistency (c.f. ATMS *NoGood* [1]):

$$P(v_1,\cdots,v_n) \wedge \mathit{active}{:}v_j \rightarrow \bot$$

where $\bot$ represents false.

RN constraints are particularly useful for expressing an incompatibility between variables without knowing if the variables will ever be active. This situation often arises when a given DCSP can be incrementally modified by simply adding a few other variables to the initial set.

4.4 Always Require Not

Analogous to the always require constraint, the *always require not* (ARN) constraint extends the basic notion of a require not constraint to rule out a variable's activity based on the activity of other variables, independent of their current value. ARN constraints have the form:

$$v_1 \wedge \cdots \wedge v_n \underset{\text{ARN}}{\Longrightarrow} v_j \qquad (\text{where } v_j \notin \{v_1,\cdots,v_n\})$$

and are logically equivalent to

$$\mathit{active}{:}v_1 \wedge \cdots \wedge \mathit{active}{:}v_n \rightarrow \neg \mathit{active}{:}v_j$$

or the corresponding NoGood.

5 Examples

The DCSP framework is being applied to two independent research problems. The first is concerned with configuration and automated design tasks [6; 14]. The second is concerned with composing the most appropriate model of a physical system given some query [4; 5]. In this section, we demonstrate our approach on some simplified examples taken from each.

5.1 Configuration

The following simple example of a DCSP has been formulated from a car configuration task (adapted from [13]). There are eight variables, of which three are in the initial variable set. Notice that the activity constraints have been used to represent many different kinds of domain knowledge such as packaging, functional requirements, structural decomposition, and incompatibilities. Similarly, compatibility constraints represent functional, structural, and packaging concerns.

Variable	Domain	
Package	{luxury, deluxe, standard}	*Initial var*
Frame	{convertible, sedan, hatchBack}	*Initial var*
Engine	{small, med, large}	*Initial var*
Battery	{small, med, large}	
Sunroof	{sr1,sr2}	
AirConditioner	{ac1,ac2}	
Glass	{tinted,not-tinted}	
Opener	{auto,manual}	

Activity constraints

1. Package=luxury $\underset{\text{RV}}{\Longrightarrow}$ Sunroof
2. Package=luxury $\underset{\text{RV}}{\Longrightarrow}$ AirConditioner
3. Package=deluxe $\underset{\text{RV}}{\Longrightarrow}$ Sunroof
4. Sunroof=sr2 $\underset{\text{RV}}{\Longrightarrow}$ Opener
5. Sunroof=sr1 $\underset{\text{RV}}{\Longrightarrow}$ AirConditioner
6. Sunroof $\underset{\text{ARV}}{\Longrightarrow}$ Glass
7. Engine $\underset{\text{ARV}}{\Longrightarrow}$ Battery
8. Opener $\underset{\text{ARV}}{\Longrightarrow}$ Sunroof
9. Glass $\underset{\text{ARV}}{\Longrightarrow}$ Sunroof
10. Sunroof=sr1 $\underset{\text{RN}}{\Longrightarrow}$ Opener
11. Frame=convertible $\underset{\text{RN}}{\Longrightarrow}$ Sunroof
12. Battery= small & Engine=small $\underset{\text{RN}}{\Longrightarrow}$ AirConditioner

Compatibility constraints

13. Package=standard $\rightarrow$ AirConditioner$\neq$ac2

14. Package=luxury → AirConditioner≠ac1
15. Package=standard → Frame≠convertible
16. Opener=auto & AirConditioner=ac1 → Battery=med
17. Opener=auto & AirConditioner=ac2 → Battery=large
18. Sunroof=sr1 & AirConditioner=ac2 → Glass≠Tinted

For this problem the smallest solutions have only four variables (Package, Frame, Engine, and Battery) and the largest have all eight. The description of a car configuration knowledge base given above would be augmented by additional constraints that represent user requirements before solving the problem. Thus, notice that changing the set of initial variables (e.g., by making `Sunroof` another initial variable) creates a somewhat different problem with different solutions, which may not simply be subsets of the solution set of the original problem.

5.2 Model composition

Compositional modeling is a method for reasoning about modeling assumptions and automatically composing the most appropriate model of a physical system for a given analytic query [4; 5]. The model composition problem is to synthesize the simplest model that is relevant to the needs of the task and consistent with the operating conditions of the system. This model must contain the parameters of interest, be able to show specified information about those parameters at a specified degree of accuracy, and minimize "cost" by reducing extraneous detail and computational effort. Many choices must be made, including the parts of the system to consider, their structural decomposition, the phenomena to consider, and how they should be modeled (e.g., what approximations can be applied).

In the compositional modeling framework, the system's models of various domains consists of a set of *elementary domain models*, each describing some fundamental piece of the domain's physics, such as processes (e.g., liquid flow), devices (e.g., transistor), and objects (e.g., container). Each elementary model is conditioned on a set of modeling assumptions stating their range of applicability and underlying approximations. Reasoning focuses on choosing among the set of possible modeling assumptions, which enable a corresponding set of elementary models, rather than reasoning about each elementary model individually. Model composition consists of four steps:

1. *Query analysis.* A query specifies a set of parameters of interest. Thus, a minimal requirement is that all of these parameters be modeled. Additionally, these parameters directly suggest further constraints. For example, a query about the level of liquid in a container indicates that a macroscopic, "contained fluids" view is called for as opposed to a microscopic "particle dynamics" view.
2. *Object expansion.* The query directly indicates a set of objects to consider, but additional objects may need to be considered to capture all relevant interactions.
3. *Candidate completion.* Some choices of simplifying assumptions raise new choices in turn. For example, considering liquid flowing through a pipe requires a decision about whether to model the fluid as compressible or incompressible.
4. *Candidate evaluation and selection.* Finally, each candidate is evaluated and the "best" candidate is selected.

Choices made during the first two stages are fully determined by the query. The DCSP framework is being used to express and solve the third stage, in which additional choices made relevant by the query must be made. In describing the possible modeling assumptions for a domain, some collections of assumptions represent mutually exclusive, alternative ways to model the same aspect of an object or phenomenon. To represent this important relationship, some assumptions are organized into sets called *assumption classes.* Each assumption class has a name c_i representing a DCSP variable; its domain is the set of assumptions in that class. The analytic query entails a set of minimal required modeling assumptions, which identify V_I, a DCSP's initial variables. The dynamic constraint satisfaction task is to extend this initial set to identify a set of modeling assumptions corresponding to a coherent and parsimonious model.

For example, consider the task of determining an appropriate set of modeling assumptions for analyzing the flow of water through a pipe connecting two containers. There are 7 variables, with the initial variable set consisting of V_I ={Precision, Can-Geometry}.[2]

Assumption Class	Domain
Precision	{qualitative,quantitative}
Can-Geometry	{quantity(level), ¬quantity(level)}
Can-Capacity	{finite-cans, infinite-cans}
Ontology	{energy-flows, contained-fluids, particle-dynamics}
Fluid-Density	{incompressible, compressible}
Fluid-Viscosity	{inviscid, viscous, non-newtonian}
Flow-Structure	{laminar, turbulent}

Activity constraints

Can-Geometry $\overset{\text{ARV}}{\Longrightarrow}$ Can-Capacity
Fluid-Viscosity=viscous $\overset{\text{RV}}{\Longrightarrow}$ Flow-Structure
Fluid-Viscosity=inviscid $\overset{\text{RN}}{\Longrightarrow}$ Flow-Structure
Precision=quantitative & Ontology=contained-fluids $\overset{\text{RV}}{\Longrightarrow}$ Flow-Structure
Ontology=contained-fluids $\overset{\text{RV}}{\Longrightarrow}$ Fluid-Density
Ontology=particle-dynamics $\overset{\text{RN}}{\Longrightarrow}$ Fluid-Density
Ontology=particle-dynamics $\overset{\text{RN}}{\Longrightarrow}$ Can-Capacity

Compatibility constraints

Can-Geometry=quantity(level) → Ontology=contained-fluids
Flow-Structure=turbulent → Fluid-Viscosity≠inviscid

[2] A more sophisticated set of representations is used in the modeling work than shown here. We have highly simplified the representation descriptions in an effort to gain clarity.

Can-Capacity=finite-cans
→ Can-Geometry=quantity(level)

Given a query about the changing levels of the two containers, these constraints elaborate the query as needed to ensure that the resulting model is coherent. For example, asking about the changing fluid levels requires a model that represents a macroscopic view of liquids and their containers.

6 Algorithm and Implementation

6.1 ATMS-based Implementation

We have implemented the dynamic CSP framework as a specialized problem solver integrated with an *assumption-based truth maintenance system* (ATMS) [1]. The algorithm is designed to use our specialized language and is summarized in Figure 1. It enables constraints about variable activity (the *problem space*) to interact with constraints about variable values (the *solution space*), producing the two problem solving levels shown in Figure 2.

The algorithm consists of a main *choose / propagate* cycle based largely on [2]. Each choose step selects an active, unassigned variable and assigns it a value that has not been previously ruled out. Each propagate step checks the constraints relevant to the new variable value assignment and propagates their consequences and dependencies. Constraint checking is ordered to take advantage of the differing scope of each constraint type. ARV and ARN constraints are checked first because they have the largest scope (i.e., they apply to variable activity, encompassing all their possible value assignments). RV and RN are checked second, because they affect a variable's activity, independent of its possible values. Finally, compatibility constraints are examined to see if the new variable value assignment is consistent. A constraint is "run" when it becomes active. Due to ATMS caching, each constraint need only be run once. Constraints that never become active during the course of problem solving are never checked. Search for each solution terminates when all active variables have been consistently assigned a value (i.e., there are no more variables for the choose step). Other variables do not appear in the solution.

Here we informally show that the algorithm is sound, which requires showing that each solution generated meets the criteria specified in section 3. The first criterion, that each solution satisfies all the constraints in $C^C \cup C^A$, is easy to show since each time a value is assigned to an active variable, all the constraints activated by that assignment are checked before the next choose.

The second criterion states that each solution must be minimal in the sense that there should be no other solution that is a superset of another. Since a solution is described by a set of variables and their associated value assignments, a proof of minimality requires proving two properties for each solution. One, no variable is assigned a value unless it is active. Two, a variable

Figure 1: Dynamic constraint satisfaction algorithm.

procedure DCSP (V_I)

$V \leftarrow V_I$
SOLUTION ← *empty*
Check all applicable activity and compatibility constraints
if CONTRADICTION(V, SOLUTION),
  then **return** *fail* (initial problem statement is inconsistent)
BACKUP? ← *false*

while V is not empty
  if BACKUP? or CONTRADICTION(V, SOLUTION)
    then backtrack and change V and SOLUTION.
      if backtracking fails,
        then **return** *fail.*
      BACKUP? ← *false*

  else if there is an active ARV constraint c_i
    then run c_i and add newly activated variables to V
  else if there is an active ARN constraint c_i
    then run c_i
  else if there is an active RV constraint c_i
    then run c_i and add newly activated variables to V
  else if there is an active RN constraint c_i
    then run c_i
  else if there is an active compatibility constraint c_i
    then run c_i
  else v_i ← SELECT&DELETE(V) {*choose next variable*}
    value(v_i) ← CHOOSE(v_i) {*choose an assignment*}
    if value(v_i) is NULL,
      then BACKUP? ←*true*
      else add value(v_i) to SOLUTION

return SOLUTION

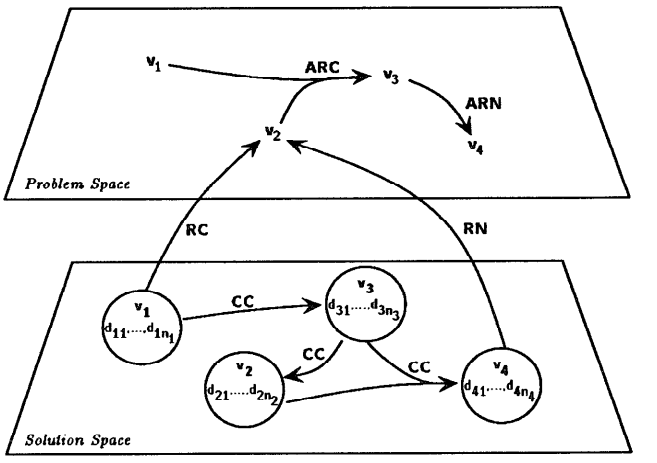

Figure 2: A dynamic constraint satisfaction network.

is made active if and only it has well-founded support from the initial set of active variables. It is easy to see that the first property holds since the only variables that are assigned values are those that have been made active by a previous step in the main loop. The second property, i.e., well-founded support for active variables can be seen to hold for our algorithm based on the following observations. *One*, at any time through the loop in the algorithm, a new variable v is made active if and only if there are some active activity constraints that justify it and there is no active activity constraint that contradicts v's activity. This ensures that a variable becomes active if and only if it has well-founded support. In other words, all active variables in a solution are required either by the initial variables or by explicitly chosen value assignments. *Two*, the only place where a choice is made (i.e., a branch in the search space) is in the assignment of a value to a variable. Each such choice generates a potentially different solution. Notice that in the language described so far, there is no disjunction over required variable constraints. While such disjunctive activity constraints will be necessary for expressing some types of domain knowledge, finding minimal solutions in the presence of such constraints can be very expensive. We are exploring heuristics that will work in some special cases. Also see [15] for an alternative formulation of disjunctive require constraints in some cases.

6.2 Example Trace

Here we briefly show a partial trace of our algorithm solving the car configuration example described in section 5.1. The algorithm does not commit to any heuristics for selecting the next variable or choosing a value for the selected variable. For this trace assume that a "smallest domain size" heuristic is used to select variables and the "first value" heuristic is used to choose variable values. We show the active variables and value assignments in bold font, constraint propagation and value choices in italics, and explanatory comments in roman. We have also shortened the names of the variables to the first letter.

$\mathbf{V_I}$ = **{P,F,E}**
C7 runs (ARV making B active)
V = {P,F,E,B}
Choose P=luxury
C1 runs (RV making S active)
V = {P=luxury,F,E,B,S}
C6 runs (ARV making G active)
V = {P=luxury,F,E,B,S,G}
C2 runs (RV making A active)
V = {P=luxury,F,E,B,S,G,A}
C9 runs, (ARV on S – no change to the active variables)
V = {P=luxury,F,E,B,S,G,A}
C14 runs (constraint between P and A, creating nogood{P=luxury, A=ac1}
Choose A=ac2
Choose F=convertible
V = {P=luxury,F=convertible,E,B,S,G,A=ac2}
C11 runs (RN on S, leading to a contradiction (C1 & C11))
Backup to the previous choice point
Choose F=sedan
V = {P=luxury,F=sedan,E,B,S,G,A=ac2}
Choose E=small
Choose B=small
V = {P=luxury,F=sedan,E=small,B=small, S,G,A=ac2}
C12 runs (RN on A, leading to a contradiction on AirConditioner activity (C2 & C12))
Backup to the previous choice point
Choose B=medium
V = {P=luxury,F=sedan,E=small,B=medium, S,G,A=ac2}
Choose S=sr1
V = {P=luxury,F=sedan,E=small,B=medium, S=sr1,G,A=ac2}
C5 runs (RV on A – no change to the active variables)
C10 runs (RN on O - no immediate effect)
C18 runs, creating nogood{S=sr1, A=ac2, G=tinted}
Choose G=not-tinted
V = {P=luxury,F=sedan,E=small,B=medium, S=sr1,G=not-tinted,A=ac2}
This is one of the possible solutions. Others can be found by backtracking.

6.3 Backtracking Implementation

A subset of our language (only the RV and CC constraints) has also been implemented by extending a conventional backtrack-search CSP framework [13]. The basic algorithm is similar to the one shown in Figure 1 with the major difference that each time an active variable is assigned a value, we use forward checking to propagate all active compatibility constraints.

This implementation was also used for making some comparisons between solving a DCSP directly using the method presented here versus solving a logically equivalent "static" CSP. The latter was obtained by introducing a new distinguished value called "null" in the domains of all variables and by appropriate transformations of all constraints such that they are trivially satisfied if one or more variables have a "null" value. In the examples that we tried, the DCSP version outperformed the "static" CSP version in all the performance metrics we measured (total time, number of backtracks, constraint checks and total variable assignments). Even on simple problems the difference in constraint checks, the metric most commonly used in the literature, was quite significant. A more detailed comparison will be presented in an extended version of this paper.

7 Discussion

In this paper we showed how to extend a constraint satisfaction framework to include constraints on activity of variables. With these extensions, we described a core language for compactly representing selection and composition constraints in synthesis tasks. We also described an algorithm that efficiently finds minimal (non-redundant) solutions to such dynamic CSPs.

We believe that the combination of the DCSP version of a problem and our method for solving such problems is effective for two reasons. One, by creating distinct

kinds of constraints, we can tune a search engine to take advantage of these distinctions in focussing the search. Two, in the DCSP version most of the variables and constraints are initially "not active". Thus, the constraint graph for the problem starts out by being very sparsely connected. It is filled only as the result of choices made by the search engine. Theoretical and empirical results [3; 8; 9] based on analyses of constraint graphs clearly show a close correspondence between the search effort and connectivity (e.g., width) of variables in the problem. The DCSP version (when relevant) "hides" many of the variables and constraints. Clearly more work is needed here, especially in terms of precise mathematical analysis that can quantify this intuition.

We are also developing improved heuristics for choosing the next active variable (e.g., most supported) or choosing a value from the domain of these variables (e.g., based on how many variables are activated by a choice). We are also looking at heuristics that help with disjunctive require constraints.

The language we have presented is only a subset of what will eventually be needed for modelling the relationships that are important in synthesis tasks. For example, referring back to the discussion in section 2, note that we need to express constraints of the form that a component may also provide additional functional roles or that it cannot be part of a configured solution by itself, i.e., without some other parent component. We have already alluded to the need for expressing disjunctive activity constraints, the simplest examples arising in cases where a component's requirements can be satisfied by disjunctive functional roles. Our methodology has been to identify language primitives for expressing some domain relationships and then examine their implications for our search method both in terms of efficiency of search and minimality of solutions found. We expect to report progress on developing a more complete language in the next paper.

8 Acknowledgments

Felix Frayman and Harley Davis made important contributions in defining the basic notion of a dynamic CSP in the context of formalizing configuration tasks. We are grateful to Dan Bobrow, Johan deKleer, Vijay Saraswat, and Mark Shirley for in-depth discussions and comments on earlier drafts of this paper.

References

[1] de Kleer, J. An assumption-based TMS. *Artificial Intelligence*, 28(2), March 1986.

[2] de Kleer, J. and Williams, B. Back to backtracking: Controlling the ATMS. In *Proceedings of AAAI-86*, August, 1986.

[3] Dechter, R. J and Pearl, J. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, December 1987.

[4] Falkenhainer, B and Forbus, K. D. Setting up large-scale qualitative models. In *Proceedings of AAAI-88*, August 1988.

[5] Falkenhainer, B and Forbus, K. D. Compositional modeling: Finding the right model for the job. (*submitted for publication*), January 1990.

[6] Frayman, F and Mittal, S. Cossack: A Constraints-Based Expert System for Configuration Tasks. In Sriram, D and Adey, R. A, editors, *Knowledge Based Expert Systems in Engineering: Planning and Design*. Computational Mechanics Publications,pp. 143–166, August 1987.

[7] Freeman-Benson, B. N., Maloney, J., Borning, A. An Incremental Constraint Solver *Communications of the ACM*, 33(1):54–63, January 1990.

[8] Freuder, E. C. A sufficient condition for backtrack-free search. *Journal of ACM*, 29(1):24–32, 1982.

[9] Freuder, E. C. A sufficient condition for backtrack-bounded search. *Journal of ACM*, 32(4):755–761, 1985.

[10] Joxan, J. and Lassez. J. Constraint Logic Programming Technical Report, IBM Thomas J. Watson Res. Ctr., Yorktown Hts., NY, USA, October 1986

[11] Mackworth, A. K. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[12] Mackworth, A. K. Constraint Satisfaction. In Shapiro, S. C. (ed.), *Encyclopedia of Artificial Intelligence*, Vol. I, 2nd edition,John Wiley & Sons, New York, 1990

[13] Mittal, S and Davis, H. Representing and solving hierarchical constraint problems. SSL Technical report, Xerox PARC, 1989.

[14] Mittal, S. and Frayman, F. Towards a Generic Model of Configuration Tasks. *Proceedings IJCAI-89*, Detroit, Michigan, August 1989.

[15] Mittal, S. Reasoning about Resource Constraints in Configuration Tasks. SSL Technical report, Xerox PARC, 1990

[16] Mulder, J., A. Mackworth, W. S. Havens Knowledge Structuring and Constraint Satisfaction: The Mapsee Approach. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 10(6):866–879, November 1988.

[17] Seidel, R. A New Method for Solving Constraint Satisfaction Problems. *Proc. IJCAI-81*, pages 338–342, Vancouver, Canada, August 24-28, 1981.