Detecting and Resolving Inconsistency and Redundancy in Conditional Constraint Satisfaction Problems

Mihaela Sabin and **Eugene C. Freuder**
Department of Computer Science
University of New Hampshire
Durham, NH 03824
mcs,ecf@cs.unh.edu

Abstract

Model debugging is an important component of assisting modelers with constraint-based problem formulation. This paper is built around a case study in modeling a special class of CSPs, which represent problems that change when certain conditions are met (Mittal & Falkenhainer 1990). The control of changing the problem, by activating or deactivating variables, is part of the problem representation and is modeled through special constraints, called activity constraints. The activity constraints may interact with the other constraints and generate inconsistencies or redundancies. We present initial examples of these two types of iteractions, and we derive more general forms of inconsistency and redundancy. We believe this work can lead to methods for automatic model debugging, which detect and resolve problems with existing models.

Introduction

Constraint Satisfaction Problems (CSPs) are simple, natural, and expressive representations of very diverse problems that may arise in many domains and can be solved effectively. However, it is indispensable first to cast these problems into CSPs in order to utilize high-performance CSP engines. Although CSP modeling is the necessary step preceding CSP solving, little work has been done on assisting modelers with constraint-based problem formulation, in particular CSP model debugging (Freuder & Huard 1993), (Keirouz, Kramer, & Pabon 1995), (Sqalli & Freuder 1998). This paper is built around a case study in modeling a special class of CSPs, known as Dynamic Constraint Satisfaction Problems (DCSPs) (Mittal & Falkenhainer 1990). The DCSPs are used to represent problems that change when certain conditions are met. The change consists in explicitly adding or removing variables, with the implicit effect of adding or removing constrains involving those variables. The case study contains initial examples of two themes: inconsistency and redundancy. From this case study we identify general forms of inconsistency and redundancy. We believe this work can lead to methods for automatic model "debugging", which detect and resolve problems with existing models. Ultimately, we are interested in *reformulating* consistent and concise models that support more efficient solution.

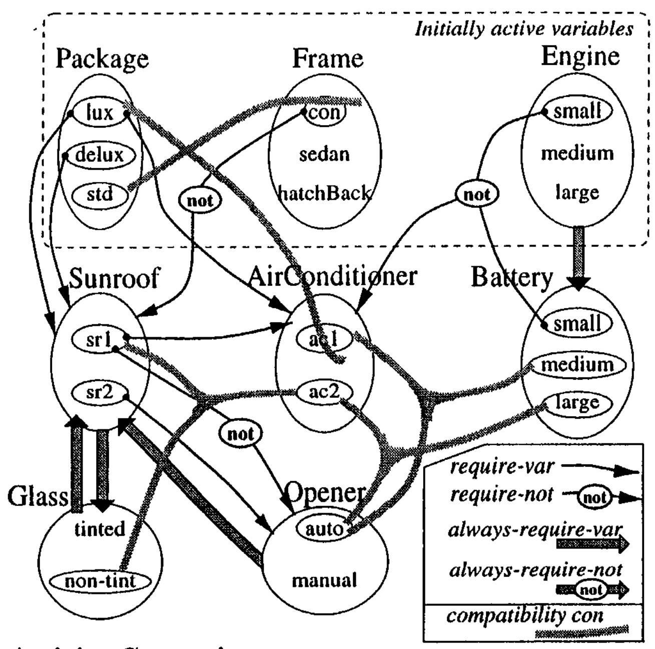
The paper is organized as follows. First, we introduce the concept of DCSP as defined by Mittal and Falkenhainer, and we present their DCSP formulation for a car configuration example. Although small and simple, the car configuration model has "bugs": conflicting constrains and superfluous constraints. In the next two sections we describe these two categories of problems with the DCSP models, inconsistency and redundancy, and we propose methods for automatic detection and resolution. In the last section, we conclude with a brief summary, and outline some directions for future work.

Background

Dynamic Constraint Satisfaction Problems

DCSPs represent problems that change dynamically during search as new variables are added to or removed from the search space. Mittal and Falkenhainer introduced the concept of *active variables* for the newly added variables, to differentiate them from the variables which are not yet part of the problem. The control of changing the problem, by activating variables, is part of the problem representation and is modeled through special constraints. These constraints are called *activity constraints* and specify conditions under which the problem changes. The conditions add or remove variables if either some variables are already active, or specific value assignments hold.

Modeling problem change into the problem distinguishes dynamic CSPs as formalized by Mittal and Falkenhainer from another class of dynamic CSPs, as first introduced in (Dechter & Dechter 1988). In the latter category of DCSPs, change occurs independently of the initial problem statement. The main focus in those DCSPs for which change conditions are not part of the original problem is on avoiding replication of work for solving the changed problem when the solution to the initial problem is known. The more integrated approach of Mittal and Falkenhainer provides additional support for reasoning about problem change. We propose to rename the DCSPs class defined in (Mittal & Falkenhainer 1990) to *Conditional Constraint Satisfaction Problems (CSPs).* This way we highlight the nature of the mechanism that changes the problem, i.e.,



Activity Constraints

A_1. Package=**luxury** $\rightarrow$ Sunroof
A_2. Package=**luxury** $\rightarrow$ AirConditioner
A_3. Package=**deluxe** $\rightarrow$ Sunroof
A_4. Sunroof=**sr2** $\rightarrow$ Opener
A_5. Sunroof=**sr1** $\rightarrow$ AirConditioner
A_6. Sunroof $\rightarrow$ Glass
A_7. Engine $\rightarrow$ Battery
A_8. Opener $\rightarrow$ Sunroof
A_9. Glass $\rightarrow$ Sunroof
A_{10}. Sunroof=**sr1** $\not\rightarrow$ Opener
A_{11}. Frame=**convertible** $\not\rightarrow$ Sunroof
A_{12}. Battery=**small** & Engine=**small** $\not\rightarrow$ AirConditioner

Compatibility Constraints

C_{13}. Package=**standard** $\Rightarrow$ AirConditioner $\neq$ **ac2**
C_{14}. Package=**luxury** $\Rightarrow$ AirConditioner $\neq$ **ac1**
C_{15}. Package=**standard** $\Rightarrow$ Frame $\neq$ **convertible**
C_{16}. (Opener=**auto**, AirConditioner=**ac1**) $\Rightarrow$ Battery=**medium**
C_{17}. (Opener=**auto**, AirConditioner=**ac2**) $\Rightarrow$ Battery=**large**
C_{18}. (Sunroof=**sr1**, AirConditioner=**ac2**) $\Rightarrow$ Glass $\neq$ **tinted**

Figure 1: DCSP formulation for the car configuration example

those conditions that trigger change through the activity constraints.

Example

Our case study is based on an example from (Mittal & Falkenhainer 1990) that illustrates a very simple car configuration task. It consists of eight variables (Package, Frame, Engine, Sunroof, AirConditioner, Battery, Glass, and Opener), 12 activity constraints (A_1 to A_{12}), and six compatibility constraints (C_{13} to C_{18}). Figure 1 shows the constraint network where both the activity control and variable compatibility are represented as edges (or hyperedges in case of constraints of arity greater than 2), and variables with their values domains are represented as nodes.

For example, variable Package has {**luxury, deluxe, standard**} as values, and activates variable Sunroof if instantiated to the value **deluxe** (according to A_3), and both Sunroof and AirConditioner variables if instantiated to the value **luxury** (according to A_1 and A_2). A *require-not-variable* constraint is A_{11}, which shows that Sunroof can not be part of a solution in which Frame takes the value **convertible**. Another type of activity constraint, *always-require-variable*, is exemplified by A_7, which makes Battery active whenever Engine is active, independent of the value assigned to Engine. A last example is the binary *compatibility constraint* C_{13}. It is defined on Package and Frame, and allows **standard** at Package while disallowing **convertible** at Frame.

The notational conventions in the statement of the activity constraints (as written to the left of the constraint network in Figure 1) are: single arrow $\rightarrow$ used for *require-variable* (RV) constraints and *always-require-variable* (ARV) constraints. A crossed single arrow $\not\rightarrow$ indicates a *require-not* (RN) or *always-require-not* (ARN) constraint. The compatibility constraints in Mittal and Falkenhainer's example are formulated as logical implications, and we use the double arrow $\Rightarrow$ for them. The constraint network in Figure 1 has attached the description of the graphical conventions for drawing the different types of edges or hyperedges that represent the problem constraints. The dotted box at the top of the constraint network marks the variables Package, Frame, and Engine as the only initially active variables.

Inconsistency

The Problem

Inconsistency may occur when both activation and deactivation of the same variable are part of the CCSP specification. The condition of the activity constraint that dictates the activation of some variable cannot hold true at the same time with the condition of another activity constraint that deactivates the same variable.

Detection

Consider the following two conflicting activity constraints:

$$A_1(\mathsf{X}): \mathsf{X} = \mathsf{a} \rightarrow \mathsf{Z}$$
$$A_2(\mathsf{Y}): \mathsf{Y} = \mathsf{b} \not\rightarrow \mathsf{Z}$$

The detection of this type of inconsistency is linear in the number of activity constraints. As each activity constraint is inspected, the activated/deactivated variable is marked with the type of activity imposed on it. If conflicting activation decisions are recorded at some variable, the detection procedure returns the activity constraints causing the inconsistency.

Resolution

One way to resolve the inconsistency problem is to infer a redundant compatibility constraint that explicitly rules out the value assignments of the two conditions, i.e., $C_{\mathsf{X},\mathsf{Y}} = (\mathtt{a}\ \mathtt{b})$. If such a constraint is not present, then the search procedure should detect the inconsistency, otherwise the inconsistency problem persists and accounts for an incorrect CCSP solution.

We illustrate the proposed resolution method using the car configuration example in Section . In our example, the activity constraints

$$A_1(\mathsf{Package}) : \mathsf{Package} = \mathtt{luxury} \rightarrow \mathsf{Sunroof}$$
$$A_{11}(\mathsf{Frame}) : \mathsf{Frame} = \mathtt{convertible} \not\rightarrow \mathsf{Sunroof}$$

introduce an inconsistency between Package and Frame, i.e., the (luxury convertible) value assignment. In addition, the activity constraints:

$$A_3(\mathsf{Package}) : \mathsf{Package} = \mathtt{deluxe} \rightarrow \mathsf{Sunroof}$$
$$A_{11}(\mathsf{Frame}) : \mathsf{Frame} = \mathtt{convertible} \not\rightarrow \mathsf{Sunroof}$$

introduce another inconsistency between the same variables, i.e., the (deluxe convertible) value assignment. The following redundant compatibility constraint enforces the detection of the conflicting activity constraints at search time. It expresses the disallowed value pairs for variables Package and Frame:

$$C^{dis}(\mathsf{Package}, \mathsf{Frame}) = \{(\mathtt{luxury\ convertible})\ (\mathtt{deluxe\ convertible})\}$$

The car example already contains a compatibility constraint between Package and Frame:

$$C_{15}(\mathsf{Package}, \mathsf{Frame}) : \mathsf{Package} = \mathtt{standard} \Rightarrow \mathsf{Frame} \neq \mathtt{convertible}$$

This constraint excludes the value assignment (standard convertible) for {Frame Package}, and, therefore, is equivalent with:

$$C_{15}^{dis}(\mathsf{Package}, \mathsf{Frame}) = \{(\mathtt{standard\ convertible})\}$$

Extending this constraint with the tuples of the redundant constraint that eliminates the activity control conflict, we obtain:

$$\begin{aligned} C_{15}^{dis}(\mathsf{Package}, \mathsf{Frame}) &= \{(\mathtt{standard\ convertible}) \\ &\quad (\mathtt{luxury\ convertible}) \\ &\quad (\mathtt{deluxe\ convertible})\} \\ &= Dom_{\mathsf{Package}} \times \{\mathtt{convertible}\} \end{aligned}$$

We see that value convertible for Frame is inconsistent with all possible values in the domain of the variable Package.

Two fixes to this problem are: (1) either value convertible is eliminated from Frame, or (2) variable Package is deactivated when Frame is convertible. In the first case convertible can not participate in any solution to the problem. The question then is whether the modeler intends to throw away this value. In a configuration task there is no use in listing a configuration feature that is forbidden to the user. In the second case a convertible Frame is incompatible with any Package option. This contradicts the status of always active of the variable Package. Thus, neither of the immediate corrections to the inconsistency problem is acceptable.

Resolving the inconsistency by explicitly maintaining tuples that disallow both activating and deactivating the same variable is straightforward. However, in some cases constraining the CSP further may lead to throwing away features or combinations of features in which the user might be interested. Apart from indicating that convertible feature can not be used in any configuration, or that there is no Package for this feature, how one can change the model to accommodate the user's requirement for convertible, in the presence of some Package option? One solution would be to add a special "luxury" feature, such as convertibleLX, as a new value for Package that goes with convertible for Frame. The challenge here is not to detect and resolve, but to correct the inconsistency problems in such a way that the model does not restrict the user selections.

Redundancy

The Problem

Redundant constraints are not always saving search effort. For example, if a constraint allows all the possible value assignments of the variables on which it is defined, none of the constraint tuples has to be checked. The absence of such constraint, in fact, eliminates unnecessary checking and saves storage space. Moreover, model maintenance and solution explanation should become easier. This particular type of redundant constraints is of no interest for the standard CSPs. It makes, however, an interesting theme in the CCSP framework.

CCSPs change during search by varying the set of active variables. Consequently, not all variables in the CCSP statement are part of the solution space. Also, at any moment during search only the compatibility constraints which are defined on currently active variables are checked. If some of the variables involved in a compatibility constraint are not active then the constraint is trivially true. The issue here is whether the CCSP has compatibility constraints which always hold, that is, they involve variables which can not be active along the same search path.

An example of a redundant constraint in the car example in Figure 1 is the C_{13} compatibility constraint:

$$C_{13}(\mathsf{Package}, \mathsf{AirConditioner}) : \mathsf{Package} = \mathtt{standard} \Rightarrow \mathsf{AirConditioner} \neq \mathtt{ac2}$$

This constraint restricts the value assignments of the variables Package and AirConditioner, and can be rewritten as:

$$C^{dis}(\mathsf{Package},\ \mathsf{AirConditioner}) = \{(\mathtt{standard},\ \mathtt{ac2})\}$$

However, if Package = standard then AirConditioner cannot be activated. Only for the other two values luxury and deluxe at Package, variable AirConditioner can be either directly (A_2)or indirectly (A_3, A_5) activated. That is, either

A_2(Package) : Package = luxury → AirConditioner
A_3(Package) : Package = deluxe → Sunroof
A_5(Sunroof) : Sunroof = sr1 → AirConditioner

Thus, $C^{disallowed}$(Package, AirConditioner) is trivially satisfied since AirConditioner cannot be activated. Eliminating the only disallowed pair for this constraint, renders C_{13} unnecessary.

Resolution

A compatibility constraint turns superfluous if it involves variables which cannot be active at the same time regardless of their instantiations. Also, a compatibility constraint tuple becomes unnecessary if its values do not belong to any of the search instantiation paths. If all the tuples in a compatibility constraint are superfluous then the constraint itself is unnecessary. C_{13} falls in the latter category. Instantiating Package with standard, only Frame, Engine, and Battery become part of the search space. All the other variables are not active for any of the instantiation paths branching out from Package bound to standard.

Once detected, the unnecessary tuples of the compatibility constraints can be simply eliminated. Entire compatibility constraints can be thrown away in this manner. The resolution procedure is straightforward and is called each time the detection procedure finds redundant tuples. Therefore, our focus is on how this type of "noisy" redundancy can be detected.

Detection

A direct way to detect the superfluous tuples of the compatibility constraints is through search. For configuration tasks that use the CCSP framework, search is performed as many times as customers request specific configuration variants. Therefore, it is worthwhile "learning" from finding all solutions to configuring a product in order to reformulate the CCSP model and save search effort for subsequent configuration requests.

The detection procedure based on searching for all solutions marks all tuples of the compatibility constraints which participate in satisfying or violating those constraints. A tuple is left unmarked if its checking does not amount for either the true or false return value of the constraint to which the tuple belongs. This means that none of the search paths contains the tuple value assignment, and thus some of the tuple variables, if not all, are inactive for all tuple instantiations during search. The tuples whose variables can not be simultaneously active can be eliminated.

Conclusion

In this paper we addressed model "debugging" of a special class of CSPs, formalized by Mittal and Falkenhainer and known as Dynamic CSPs. These DCSPs change dynamically as search progresses based on the activity control defined for the problem variables. The problem change is encoded in the DCSP statement through specialized constraints, which describe *conditions* that can trigger variable activation or deactivation. Since other DCSP frameworks do not incorporate problem change into the CSP model, we decided to make this distinction clear, and renamed Mittal and Falkenhainer's DCSP to Conditional CSP.

We proposed to detect and resolve two types of problems with CCSP modeling: inconsistency and redundancy. Inconsistency problems occur when a variable is both activated and deactivated along a search path. Detection in this case records the conflicting activity constraints. The resolution procedure infers compatibility constraints that rule out value assignments that lead to inconsistent variable activity. Note that the inferred compatibility constraints save search effort by signaling the inconsistencies prior to checking all the activity constraints.

The inferred constraints that solve activation inconsistency add "positive" redundancy to the CCSP formulation. There are, however, other redundant constraints, unnecessary or superfluous, that do not affect positively the search effort. These constraints create a "noisy" redundancy, with which we associated the second source of problems with CCSP modeling. Unnecessary compatibility constraints involve variables which cannot be all active for any of their variable instantiations. Other compatibility constraints can be partially superfluous. This is the situation where some of the constraint tuples define value assignments for which the involved variables cannot be simultaneously active. The resolution procedure simply eliminates the unnecessary tuples and constraints. The detection mechanism we outlined in this paper is an add-on to searching for all solutions. We are interested to find out if model parsing is enough for the redundancy detection task.

There are several interesting directions emerging from this work that need further investigation. In particular, the interaction between the activity and compatibility constraints should be examined more carefully. Are there other types of inconsistency and noisy redundancy apart from the ones discussed in this paper? Some answers are based on the special set of always active variables. For example, deactivating such a variable, either directly or indirectly leads to an inconsistency, while activating it adds noise to the activity control. Unnecessary redundancy occurs also if deactivating a variable is doubled by constraining incompatibility with that variable through compatibility constraints. Finally, understanding the "tension" between the activity and compatibility constraints is of particular interest when proposing static CSP-based reformulations for the CCSPs. These reformulations benefit indeed from the extensive work on static CSP search and inference. Yet they may hide useful information in the original CCSP.

Acknowledgments

This material is based on work supported by Trilogy and the National Science Foundation under Grant No. IRI-9504316, and it was presented at the CP'98 Workshop on Constraint Problem Reformulation, October 1998, Pisa, Italy.

References

Dechter, R., and Dechter, A. 1988. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, 37–42.

Freuder, E., and Huard, S. 1993. A debugging assistant for incompletely specified constraint network knowledge bases. *International Journal of Expert Systems: Research and Applications* 419–446.

Keirouz, W.; Kramer, G.; and Pabon, J. 1995. *Principles and Practice of Constraint Programming.* Cambridge, MA: The MIT Press. chapter Exploiting Constraint Dependency Information for Debugging and Explanation, 183–196.

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the 8th AAAI)*, 25–32.

Sqalli, M., and Freuder, E. 1998. Integration of csp and cbr to compensate for incompleteness and incorrectness of models. In *AAAI-98 Spring Symposium on Multimodal Reasoning.*