straints which conflict with them are activated. Thus, in example a) of Figure 5, new values for feasible and preferred intervals of `Pier-width` are found by propagating the constraint from `Beam-depth`.

Preferences represent optimal values, and the closer the value is to the preference the better the criterion is adhered to. When preferences are in conflict, they are therefore not dropped, but *weakened* so that the preferred value becomes the feasible value closest to the interval defined by the preference constraint. Thus, in example b) of Figure 5, the preferred interval of `Pier-width` is now set to [2.4..2.4], an interval representing the value which is closest to the interval defined by the preference constraint. In a later propagation step, this single preferred value may propagate to also fix a single value for `Beam-depth`, as shown in the figure.

In both examples of Figure 4 and Figure 5, the label of `Beam-depth` stays intact even after conflict resolution. This assumes that it does not depend on the values for `Pier-height` and `Pier-width`. If there is such a dependency, our system follows the truth maintenance algorithm of Doyle ([DOY79]) and withdraws the support of all consequences of the revised variables. In the example, the resulting contexts would then be identical to the ones shown, but lack all assertions about the variable `Beam-depth` which would have to be rededuced.

5 Conclusions

We have presented an algorithm for dynamic constraint satisfaction with continous variables, an important problem which has not been addressed in the literature. We have shown that the nature of the problem in continous domains is different from discrete domains. In particular, the unbounded nature requires an *incremental* construction and solution of the constraint network. Contrary to common belief, few published techniques for constraint propagation are applicable to continous domains, and we hope that the topic will receive more attention by the research community in the future.

The incremental solution means that the final results depends essentially on the order in which solutions are generated. Our algorithm provides defaults and preferences as a means to specify heuristics for controlling this ordering. This control is important in view of the fact that realistic design problems are too complex to model domain constraints completely and accurately, and heuristics are necessary to ensure that the result is actually useful.

The algorithm we have developed has been implemented as a central element in an intelligent CAD system for civil engineers. The mechanisms it provides allow us to reproduce the decision making in civil engineering designs such as that of Figure 1. We are currently working on improving the system's knowledge base, and applying the DCSP algorithm to problems in other design domains such as architecture and mechanical engineering.

Acknowledgements

We thank S. Boulanger, G. Kimberley and M. Jordi, ICOM, EPFL, for advice on practical aspects of civil engineering, H. Kefeng, LIA for many of the early ideas of this project, and the Swiss National Science Foundation for sponsoring this research under contract No. 20-27591.89

References

[DAV87] **E. Davis**: "Constraint Propagation with Interval Labels," *Artificial Intelligence* **32**, 1987

[DD88] **R. Dechter, A. Dechter**: "Belief Maintenance in Dynamic Constraint Networks," *Proceedings of the 7th National Conference of the AAAI*, St. Paul, 1988

[DKL84] **J. de Kleer**: "An Assumption-based Truth Maintenance System," *Artificial Intelligence* **28**, 1986

[DP87] **R. Dechter, J. Pearl**: "Network-based Heuristics for Constraint Satisfaction Problems," *Artificial Intelligence* **34**(1), 1987

[DOY79] **J. Doyle**: "A Truth Maintenance System," *Artificial Intelligence* **12**, 1979

[FR82] **E. Freuder**: "A Sufficient Condition for Backtrack-Free Search," *Journal of the ACM* **29**(1), 1982

[MA77] **A. Mackworth**: "Consistency in Networks of Relations," *Artificial Intelligence* **8**, 1977

[MO74] **U. Montanari**: "Networks of Constraints: Fundamental Properties and Applications to Picture Processing," *Inform. Scie.* **7**, 1974

[MF90] **S. Mittal, B. Falkenhainer**: "Dynamic Constraint Satisfaction Problems," *Proceedings of the 8th National Conference of the AAAI*, Boston, 1990

[PE91] **C.J. Petrie**: "Context Maintenance," *Proceedings of the 9th National Conference of the AAAI*, Anaheim, 1991

[SCH86] **A. Schrijver**: "Theory of Linear and Integer Programming," Wiley, 1986

[WA75] **D. Waltz**: "Understanding Line Drawings of Scenes with Shadows," in P.H. Winston (ed.): *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975

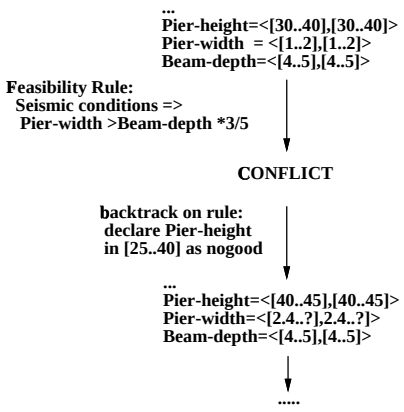

Figure 4: *Resolution of a feasibility conflict.*

When the rule condition is satisfied for only part of the preferred interval, two contexts with the corresponding feasible subintervals result. In our implementation, a heuristic rule explores the case corresponding to the larger subinterval first. Thus, in the example of Figure 3 on the left, the label $\texttt{Pier} - \texttt{height} =< [30..60], [30..45] >$ is transformed into $\texttt{Pier} - \texttt{height} =< [30..40], [30..40] >$.

After a new context has been generated, the admissible intervals of its variables are adjusted using the Waltz algorithm for arc consistency ([WA75]). Since Davis ([DAV87]) has shown that the Waltz algorithm is incomplete for continuous domains, we use the feasibility test of Fourrier & Motzkin ([SCH86]) to verify that the constraint set actually admits a solution.

4 Conflict resolution

Whenever the active constraint set in a context does not admit a consistent labelling, a conflict occurs. By retracting one of the conflicting constraints, a new and consistent context must be generated to continue problem solving. We distinguish two kinds of conflicts: *preference* conflicts and *feasibility* conflicts.

A feasibility conflict occurs when a constraint is incompatible with currently asserted feasible intervals for the variables involved. An example of a feasibility conflict which could arise as a consequence of the feasibility rule application of Figure 3 is shown in Figure 4. It is resolved by *backtracking* on the activation conditions of the constraints which influence the involved variables. This set of constraints is generated by tracing the justifications of the interval bounds which are violated, and constraints are tried in turn until the conflict is resolved. The inconsistency is recorded by a TMS as a `nogood` to ensure that it is not tried again.

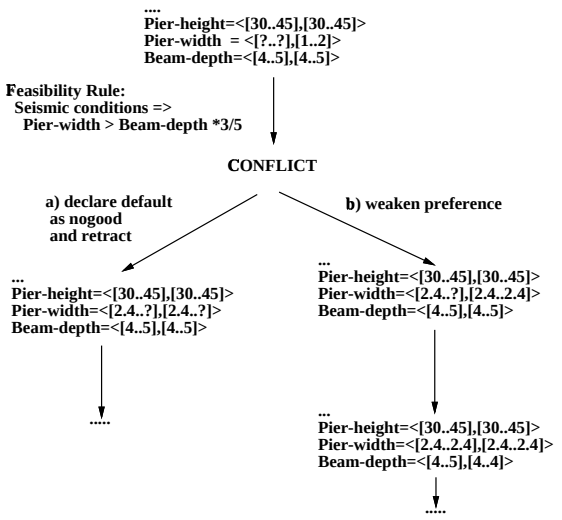

Figure 5: *Resolution of a preference conflict*

In order to explore all possible detail solutions before changing global characteristics, the system always retracts the most recently asserted feasibility constraint, provided that its justification is not an unretractable premise. In the example of Figure 4, we assume that the seismic analysis which introduces the constraint $\texttt{Pier} - \texttt{width} \geq \texttt{Beam} - \texttt{depth} * 0.6$ is such an unretractable premise, so that the system backtracks to the rule $\texttt{Pier} - \texttt{height} \in [25..40] \Rightarrow \texttt{Pier} - \texttt{width} \in [1..2]$. It declares the interval [25..40] a `nogood` for the `Pier-width`, and generates a new context in which this interval is eliminated.

A preference conflict occurs when a constraint is incompatible with the preferred intervals. It can occur either directly when a new preference or default is introduced, or indirectly when a new feasibility constraint retricts the preferred interval. An example of a preference conflict which might arise from the application of the preference rule in Figure 3 is shown in Figure 5. Similarly to feasibility conflicts, the system picks a constraint to relax by tracing the justifications of the interval bounds of the variables involved in the conflict. When there are several such constraints, the system follows the heuristic of relaxing the more general one first, since it is based on less information than the detailed default or preference. In the case where the constraint is introduced in a case split, it is relaxed by backtracking as described earlier. In all other cases, the system follows the following relaxation procedures.

Default constraints are used to make assumptions required to proceed with reasoning, and consequently are *dropped* without consequence as soon as other con-

ticular characteristic or dimension of the design. For example, in a bridge design task, the type of bridge, the number of piers, and the length of the deck are all represented by variables.

The set of admissible values of a variable is represented by its *label*. The label is a tuple of two intervals:

<[feasible interval],[preferred interval]>

- the *feasible* interval represents the limits of physical feasibility, and
- the *preferred* interval, which often collapses to a single value, represents the currently preferred values.

The preferred interval must always fall within the feasible interval, and whenever the feasible interval is changed the preferred interval is adjusted as required. When there is no more intersection between feasible and preferred interval, the preferred interval becomes the feasible interval boundary which is closest to the originally preferred one. Distinguishing feasible and preferred intervals allows us to separate control and domain knowledge: the constraints of the domain are represented by the feasible intervals, while the preferred intervals provide the basis for formulating control heuristics.

A *constraint* is an algebraic equality or inequality involving variables and constants. Variables and constraints can be either *static*, introduced as a premise to the problem, or *dynamic*, introduced by rules as a consequence of design decisions. Constraints and variables are asserted with justifications that are managed in a justification-based reason maintenance system ([DOY79]) which has been extended to accommodate dependencies between bounds of variable intervals.

3 Forward reasoning

Conceptual design is based on a set of *design rules* which introduce structures or properties. For the constraint satisfaction formulation, they result in the introduction of variables and constraints. As mentioned earlier, a DCSP with continous variables often has an unbounded number of solutions, so that no complete equivalent constraint network can be constructed. This rules out the ATMS-based solutions proposed in [MF90] and the network-based methods proposed in [DD88]. Our algorithm performs a *sequential* search of the possible constraint networks, each represented by a *context* consisting of the following elements:

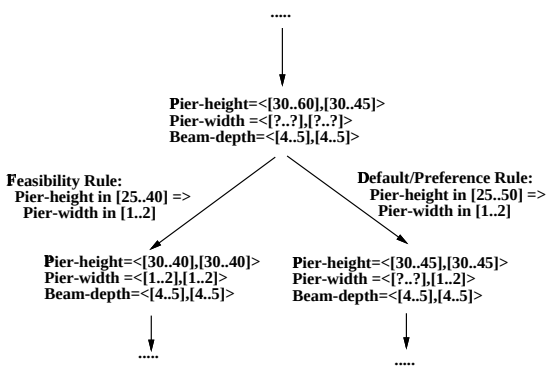

Figure 3: *Left, forward application of a feasibility rule. Right, forward application of a default or preference rule. Question marks (?) mean that the corresponding interval bound is unknown.*

- a set of variables with their associated labels
- a set of active constraints
- justifications of constraints and variable labels in terms of the decisions that introduced them.

Given that the current context is *feasible* - the set of constraints allows a solution - rules are fired which deduce a new context. Rules are ordered in *levels* of detail, and the most general rules are fired first. Our algorithm distinguishes feasibility, default and preference rules and corresponding types of constraints. Their application is illustrated in Figure 3. *Feasibility* constraints act on the feasible intervals of the involved variables, as shown on the left. *Preference* and *default* constraints affect the preferred interval only and leave the feasible interval unchanged, as shown on the right.

When a rule condition refers to the values of a variable in the current context, it is evaluated using the *preferred* interval of the variable's label. The preferred intervals thus serve the purpose of focussing the reasoning on the most promising subintervals. The distinction between preferred and feasible intervals separates control and domain knowledge, corresponding to the ideas of explicit context maintenance also proposed by Petrie ([PE91]).

Once a preferred interval has been used to activate a rule, it becomes the feasible interval of the corresponding variable in all resulting contexts. Thus, applying a rule with the condition $\texttt{Pier} - \texttt{height} \in [25..50]$ to the label $\texttt{Pier} - \texttt{height} =< [30..60], [30..45] >$ produces the label $\texttt{Pier} - \texttt{height} =< [30..45], [30..45] >$, as shown in Figure 3 on the right. This ensures that only values for which the context is actually valid will be considered in subsequent reasoning.

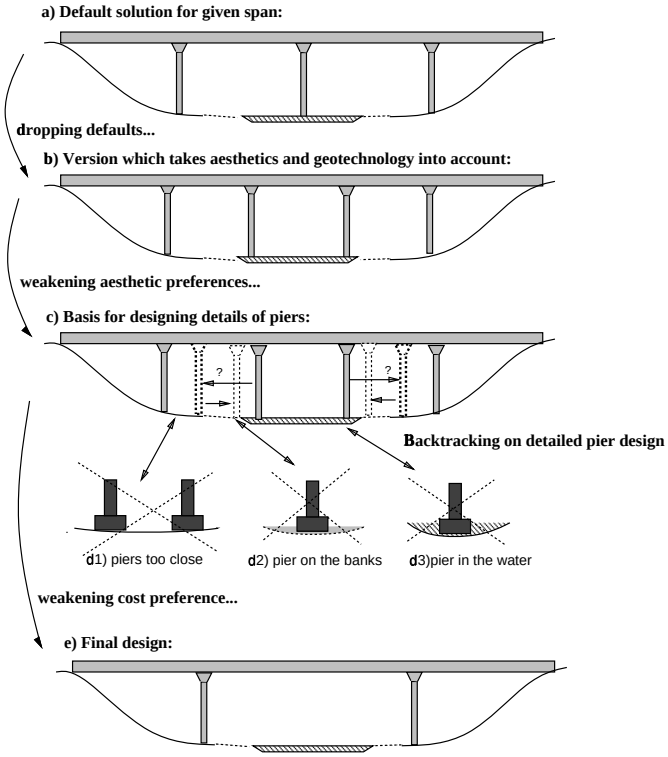

Figure 1: *Different phases in the design of a bridge, taken from the example of the French Creek bridge in British Columbia, Canada. The evolution from solution a) to d) illustrates the need for dynamic constraint satisfaction. Different reasoning strategies are involved: a) ⇒ b) involves dropping a default value, b) ⇒ c) and c) ⇒ e) weakening aesthetic and cost preferences, respectively. In developing c), the designer attempts to design piers at different locations, but is forced to backtrack because of unsatisfiable constraint combinations.*

dynamic constraint satisfaction is therefore straightforward to implement. In continuous domains, the fact that value sets cannot be enumerated makes this solution impossible. For example, a bridge can have an unbounded number of spans, and consequently the associated constraint network has unbounded size. It can only be generated and maintained incrementally, requiring constraint propagation to be coupled with an explicit reasoning process. This paper describes an algorithm for solving DCSP that combines constraint propagation with interval labels ([DAV87]) with an efficient search methodology.

Since constraint contexts are generated sequentially, it becomes important to provide *control regimes* that avoid extensive exploration of undesirable constraint combinations. Solving DCSP can involve many different reasoning strategies, which are illustrated in Figure 1. The transition from solution a) to b) implies *dropping a default* (three piers). In solution c), the position of the piers becomes variable by *weakening* aesthetic preferences on pier position. Solutions d1) through d3) are attempts to design piers for particular placements, each of which turns out to be infeasible and causes *backtracking* to solution c): in d1), the piers are too close together, in d2), the banks are too soft to support the weight, and in d3), the water flow so rapid as to cause erosion problems. Finally, solution e) is a result of *weakening* cost preferences, since physical constraints make all solutions with four piers unacceptable. Our DCS algorithm provides general methods for defaults and preferences which allow to reproduce such reasoning in a computer.

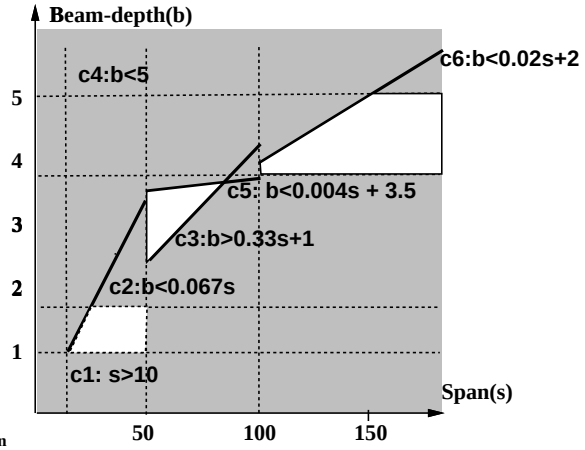

Figure 2: *In a dynamic constraint satisfaction problem, even linear constraint can result in an unbounded number of disjoint regions. In this Figure, there are three regions, bounded by {c1,c2}, {c3,c5} and {c4,c6}. Constraint propagation has to be combined with explicit search among potentially feasible regions.*

Our algorithm is the first to address the dynamic constraint satisfaction problem in continuous domains. The DCSP formulation is a more natural way to express design constraints than static formalisms. The algorithm we describe has been developed in the context of a civil engineering design project, and some of its characteristics are motivated by this domain. However, the formulation and solution we present can be applied to problems in other domains where static constraint networks are insufficient.

2 Problem formulation

A constraint satisfaction problem is defined by *variables* and *constraints*. A variable stands for a par-

Dynamic Constraint Propagation with Continuous Variables

Boi Faltings, Djamila Haroud, and Ian Smith
Artificial Intelligence Laboratory (LIA)
Swiss Federal Institute of Technology (EPFL)
IN-Ecublens, 1015 Lausanne, Switzerland

Abstract. Traditional techniques for constraint satisfaction assume that the set of constraints is independent of the results of the propagation process. In many practical problems, especially in automated design, applicability of constraints depends upon assumptions about parameter values, resulting in a *dynamic* constraint satisfaction problem.

While dynamic constraint satisfaction in discrete and finite domains is straightforward, continuous domains may result in unbounded numbers of disjoint contexts. Consequently, the complete constraint network can have unbounded size, and must be maintained incrementally together with constraint propagation. In this paper, we show an approach to solving this problem using the interval propagation algorithm of Waltz, a reason maintenance system, and the Fourrier-Motzkin algorithm for detecting conflicting constraint sets.

Keywords: Constraint Propagation, Constraint Satisfaction, Design

1 Introduction

Design can be viewed as a combination of two tasks: *conceptual design* defines the structure and its parameters, and *detail design* finds an appropriate set of values for the parameters in the solution. Conceptual design manipulates partial descriptions of the artifact, consisting of parts and properties. Parts define continuous *variables*, and properties *constraints* on these variables. In order to guide reasoning, it is important to determine whether and within what bounds such a partial description corresponds to a *feasible* physical object. In the example of Figure 1, constraint propagation guides the search so that a physically realizable structure is produced.

Mapping the symbolic representations found in conceptual design to actual objects is a *constraint satisfaction* problem. Conceptual design manipulates solution regions, whose characteristics can be determined using constraint *propagation*. Constraint propagation was first applied by Waltz ([WA75]), formalized by Montanari ([MO74]) and Mackworth ([MA77]), and further extended by a community of researchers since then. It is often used to preprocess a constraint satisfaction problem to facilitate the solution by search ([FR82, DP87]). In conceptual design, constraint propagation ensures that subsequent detail design finds a feasible solution.

Most work in constraint propagation addresses variables with discrete and finite domains. Design often requires constraint propagation techniques for *continuous* variables, as investigated by Davis ([DAV87]). Another shortcoming of classical algorithms for constraint propagation is the assumption that the set of constraints is static and therefore independent of particular solutions. In the design problem shown in Figure 1, this condition is not satisfied: going from solution a) to solution b) involves changing the number of spans and thus the set of variables and constraints. Since the change was made in response to variable values in solution a), the new variables and constraints depend upon the results determined by earlier propagation. This is an example of a *dynamic* constraint satisfaction problem (DCSP), where active constraints and variables depend on variable values within the same problem.

Figure 2 shows that in a dynamic constraint satisfaction problem, the application conditions of constraints generate a *search space* of different combinations of active constraints, which can form disjoint solution sets. Constraint propagation as well as optimization techniques generally assume a single convex solution set. In order to satisfy this assumption, solving a DCSP requires a combination of *search* among consistent constraint combinations and *propagation* of the constraints separately within each solution region.

In discrete and finite domains, conditions for existance of variables and constraints can themselves be expressed as discrete constraints (see [MF90]), and