Constraint Management for Collaborative Electronic Design

Juan Antonio Carballo
EECS Department, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109, USA
+1 (734) 936-2828
jantonio@umich.edu

Stephen W. Director
College of Engineering, University of Michigan
1221 Beal Avenue, Ann Arbor, MI 48109, USA
+1 (734) 647-7010
director@umich.edu

ABSTRACT

Today's complex design processes feature large numbers of varied, interdependent constraints, which often cross interdisciplinary boundaries. Therefore, a computer-supported constraint management methodology that automatically detects violations early in the design process, provides useful violation notification to guide redesign efforts, and can be integrated with conventional CAD software can be a great aid to the designer. We present such a methodology and describe its implementation in the Minerva II design process manager, along with an example design session.

1. INTRODUCTION

The trend towards more complex VLSI designs and increasingly tight time-to-market constraints requires ever-larger design teams, where different parts of the design (e.g., hardware and software, or product and manufacturing processes) are performed in parallel by different groups of designers. Such concurrent design exploits the inherent parallelism in the design of a large system. Unfortunately, most concurrent design methodologies suffer from the fact that conflicts involving multiple parts of the design are often detected late in the design cycle when the parts of the design are put together and tested against global specifications. Resolving these conflicts requires very costly rework.

If one views the entire design as a network of *constraints*, i.e., relations among the values of design variables, *conflicts*, then, are violations of constraints. The situation described above occurs because each group of designers typically considers only a subset of all the constraints relating the part of the system they are considering to the other parts of the system. Only when the parts are put together are all constraints considered simultaneously. If, instead, all constraints are considered while these parts are being concurrently generated, then costly rework can be avoided. The large number and variety of constraints make computer support for this task essential.

This paper presents a constraint management methodology called CCM (for **C**ollaborative design **C**onstraint **M**anagement) that provides automated support for conflict detection and resolution as an integral part of the design process. This methodology has been implemented in the Minerva II design process manager [15].

Realizing this constraint management methodology required that a number of issues be addressed, including:

- the dynamic *generation* of all constraints among the parts of the design as it is hierarchically and recursively decomposed;
- *evaluation* of all of these constraints despite the fact that they are very numerous and varied in nature, often involving multiple disciplines;
- handling the interconnectivity of the constraint network caused by many constraints affecting the same design variables; and
- avoiding designer information-overload by *notifying* only those designers affected by conflicts and presenting them only with information essential for guiding re-design efforts.

While previous work has produced techniques to incorporate data and meta-data management [3,8], flow management [13,16], network infrastructure [2], and design process management services [10,14] in electronic CAD environments, they have not directly addressed these issues due, in part, to the complexity of the knowledge needed to generate and evaluate networks of constraints of arbitrary form[1]. However, this previous work, in conjuction with work in constraint-based systems [1,11] and CAD tools, can be used as a basis upon which a suitable constraint management capability can be built.

Specifically, we can make use of the fact that existing CAD tools essentially verify designs against constraints or synthesize constraint-meeting designs, and that work in constraint-based systems [1,11] has produced efficient software that can detect conflicts and eliminate infeasible solutions in a constraint network. To take advantage of these features, we represent each constraint as an abstract function identifier, while the details of evaluating the constraint are encapsulated in the execution of a CAD tool or a constraint-based system. A constraint propagation algorithm can then be used to detect conflicts as soon as possible. Generation and evaluation of arbitrarily complex constraint networks is thereby made feasible without the need to reinvent existing methodology.

We can also take advantage of the ability that existing design process management tools have to keep track of the design history [15]. This is achieved by (1) associating with each conflicting value its design history, and (2) selecting the information to be included in the violation notification from the design history database. We then provide a mechanism to control the notification policy, including what history information is selected and which designers receive this information. The notification thereby includes only information that is useful to resolve the conflict, and information overload is prevented.

The remainder of this paper is organized as follows. Section 2 describes the CCM methodology. Section 3 describes the implementation of the methodology in the Minerva II Design Process Manager. Finally, Section 4 draws conclusions.

[1] Although not focused in electronic design, work in the area of concurrent engineering [7,9] has produced tools that help designers detect and resolve violations of some types of parametric constraints. These tools cannot generate and evaluate constraints of arbitrary form.

2. COLLABORATIVE DESIGN CONSTRAINT MANAGEMENT

2.1. Background

To facilitate the discussion of CCM, we define the necessary concepts to describe design processes and the role of constraints in these processes. These concepts are based on the design formalism defined by Jacome [10] and refined by Sutton [14].

The *design process history*, denoted by H_n, is given by (1) a sequence of state-transition pairs denoted by $\{<s_i, t_i>, i=1,...,n-1\}$, where s_i denotes the *design process state* at stage *i*, and t_i denotes the *transition* between s_i and s_{i+1}; and (2) the *current design process state* s_n. Each state s_i is composed of three components described below.

- The *design object hierarchy* is the set of all *design objects* currently under design. Objects are design artifact representations, organized by abstraction level and by decomposition relationships. Each object is described in terms of a set of design variables called *properties*, each of which represents a characteristic of the object. Making a design *decision* implies deriving a property value. The *set of values* of a property represent all the alternatives being considered by the design team for that property, and can take arbitrary forms, including numbers, strings, vectors, and complex descriptions.
- The *design problem hierarchy* is the set of all *design problems* formulated so far. A design problem is a described in terms of its *objective* (i.e., the design function to be carried out), the *object* to be designed, the target *abstraction level*, the *input properties*, and the targeted *output properties*. For example, a design problem may be formulated by combining the "Synthesize" objective, the "branch predictor unit" object, the "gate" target abstraction level, the "2 Watt maximum-power requirement" input property, and a gate-level description as a target. Each problem has a *status*, indicating its level of accomplishment (e.g., "Solved").
- The *constraint network* is a set denoted by $C_i=\{c_j, j=1,...,N_i^C\}$, where each c_j is a *constraint* and N_i^C is the number of constraints in the design. Constraints represent relations that must be met by groups of properties for the design to be correct. Each constraint c_j is given by (1) the set $A_j=\{a_k, k=1,...,N_j^A\}$ of the N_j^A properties affected by c_j; (2) a *relation*, denoted by ρ_j; and (3) a *status* indicating whether the relation ρ_j is currently met by the properties' values.

A transition t_i is a state change from s_i to s_{i+1}. Each transition t_i is caused by a *design step*, denoted by T_i. A design step is an problem-solving action taken by a designer. Fig. 1 shows an example of a transition caused by a design problem decomposition step.

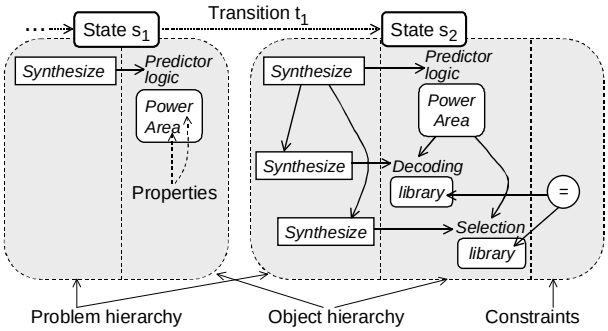

Fig. 1. A design state transition caused by the decomposition of a design problem (to "Synthesize" the "Predictor Logic") into two subproblems (to "Synthesize" the "Decoding" and "Selection" subunits).

A *designer* is a human agent capable of making design decisions. Each designer in the team is represented in the state s_i by an identifier, denoted by d_j. Objects, properties, problems and/or constraints are associated with designers through ownership or interest relationships.

2.2. Coordinating the design process

CCM is based on interleaving the design process with a *coordination process* that is executed every time a design process step is taken, as depicted in Fig. 2. The coordination process is aimed at detecting conflicts and communicating these conflicts to the designers affected by them.

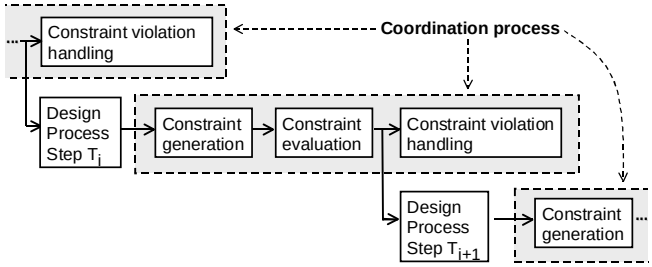

Fig. 2. The coordination process.

Based upon the chosen design step T_i, the problem and object hierarchies are updated to reflect the effect of this step (see problem and object hierarchies in Fig. 1). Then, the coordination process starts with *constraint generation*, which takes as input the state s_i and the step T_i and derives the *constraint network update* ΔC_i, i.e., the constraints and properties to be added to and/or deleted from C_i to form C_{i+1} (e.g., the "=" constraint and the "library" properties in Fig. 1). *Constraint evaluation* is performed upon completion of constraint generation. It takes as input the new network of constraints C_{i+1}, and computes the *conflict information* ξ_{i+1}, i.e., the status of the constraints in C_{i+1} and the property values that are inconsistent. When constraint evaluation is finished, the transition from s_i to s_{i+1} is complete, and the differences between s_i and s_{i+1} include the modifications due to ΔC_i and ξ_{i+1}. *Constraint violation handling* is then executed. Each new conflict represented in the new history H_{i+1} is examined in order to determine its notification policy. Conflicts are then communicated to designers by following the computed notification policies. Since constraint violation handling does not modify the design state, it can be executed in parallel with the next design step T_{i+1}[1].

2.3. Architecture

As depicted in Fig. 3, the architecture of the proposed methodology is based on a set of distributed software modules that communicate with standardized message protocols, and the designers, who communicate through a user interface.

The *design process manager* interacts with an arbitrary number of designers to help them manage the design process, supplementing the services of existing tools or frameworks. However, unlike conventional design process managers, CCM's carries out constraint generation at each step T_i and sends the result, ΔC_i, to the *design constraint manager*. This manager then handles constraint evaluation and returns the conflict information ξ_{i+1} to the design

[1] Note that modeling the design process as a step sequence does not preclude multiple designers from working in parallel. We only assume that no two designers commit a design step exactly at the same time.

process manager. The *notification manager* then carries out constraint violation handling, which ends in notification of new conflicts.

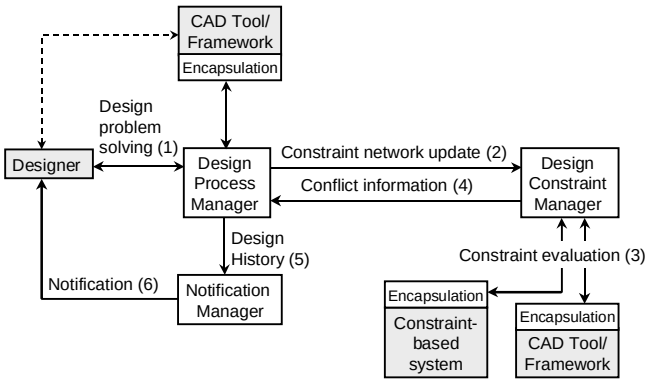

Fig. 3. Architecture of the CCM methodology.

2.4. Constraint generation

Generation of constraints from "top-level" specifications must be automated because it is complex and occurs frequently in leading-edge design processes. Creating a constraint generation engine is a difficult task as constraints vary in form and the types of constraints to be generated are discipline-dependent. We simplify this task by recognizing that constraints can be categorized into classes, and that generating a constraint of a given class does not require explicit knowledge of the form of the constraint.

In ΔC_i, the relation ρ_j of each constraint c_j is represented by an *identifier* denoting the generic "class" of constraints that c_j belongs to. For example, for the circuit board design discipline, the "totalCost" class corresponds to all constraints of the form $\sum_i cost_i = Cost_{total}$, with i=1,...,n, where n is the number of parts on a board. The constraint generation engine is simplified because the design process manager does not need to send the detailed expression of the relation to the constraint manager. As will be explained below, this identifier is enough information for the constraint manager to handle evaluation of the constraint.

In order for the design process manager to generate constraints for any given design discipline (i.e., to compute ΔC_i), it needs to have knowledge about what, and when, constraints must be introduced. We have extended *DDDL* [14], a language for describing design discipline knowledge, to provide the ability to embed constraint generation knowledge within this discipline knowledge[1].

We consider three cases when a constraint must be generated. First, for each newly introduced property, there may be a constraint on its values. Second, when a design problem is formulated, constraints may appear among the properties of the object involved in the problem. Third, when a problem is decomposed into simpler subproblems, constraints may appear among the properties of the subproblems' objects and the properties of the parent problem's object. In the first case, constraint knowledge can be embedded within *property* class knowledge; in the second, within *object* class knowledge and in the third, within *decomposition* knowledge.

An example of a constraint knowledge description in our extension of DDDL is shown below, where a description of a class of *constraints* (areaConstraint) is embedded in a description of a type of problem *decomposition*. When a designer decomposes a design function (e.g., "synthesize") applied on a processor block into the same function applied on two or more processor blocks, a constraint that relates the layouts of all subblocks and the area of the parent block is generated.

```
domain processorBlock {...
  decomposes into {
      components 2+ processorBlock;
      constraint areaConstraint {
          arguments {
              all descendant.layoutDescription;
              parent.area;
          }
          evaluation function areaRelation;
      }
  }...
}
```

The *arguments* of a constraint, i.e., the properties involved in the constraint, are selected in DDDL with a *filter*. The DDDL constructs to define this filter are loosely based on first-order logic [6] including some specialized constructs, such as PARENT (the parent object in the decomposition) and DESCENDANT (the children in the decomposition), as shown in the example.

Due to the large variety of constraint expressions, explicitly capturing these expressions in DDDL would make its syntax very complex and knowledge capture time-consuming. Instead, we replace each complex expression with an abstract function identifier called the *evaluation function* ("areaRelation" in the example), thereby encapsulating the details of these expressions. The constraint manager handles the evaluation of these functions.

2.5. Constraint evaluation

Computing the conflict information ξ_{i+1} requires finding property values or alternatives that cannot simultaneously satisfy all constraints affecting them directly or indirectly through constraint network paths. The constraint manager achieves this goal by running the constraint propagation algorithm shown below on the network C_{i+1}. The algorithm is an adaptation of existing constraint propagation algorithms [1,11] that is applicable to constraints of arbitrary form and provides the ability to leverage legacy software.

```
procedure CONSTRAINTPROPAGATION(network C_{i+1})
  Constraint queue Q ← C_{i+1}; Request queue R ← ∅ ;
  while {(Q not empty) and (R not empty)} {
      (G, E) ← SELECTCONSTRAINTGROUPANDEVALUATOR(Q);
          /* All constraints in G are selected from queue Q */
          /* and can be evaluated by constraint evaluator E */
      Q ← Q-G;
      SENDREQUESTTOEVALUATOR(G, E);
      ADDREQUESTTOQUEUE(G, R);
      CHECKIFANYEVALUATORRESPONDED();
      For each (constraint group J with a response) {
          UPDATESTATE(J, C_{i+1}); REMOVEFROMQUEUE(J, R);
          Z ← {∀ constraint c_j where c_j ∉ J and
                  ∃ a_k argument of c_j such that X_k has shrunk};
          Q ← Q+Z;
      }
  }
  return final state of C_{i+1};
```

[1] DDDL could not describe constraints of arbitrary form.

The algorithm takes as input the *initial state* of the network C_{i+1}, where each property a_k has an initial value set, denoted by X_k^I, and the status of each constraint is unknown. The algorithm deletes infeasible values from each X_k^I. At the end of the execution of the algorithm, the *final state* of C_{i+1} is returned. This state is identical to the initial state, except that (1) the value set of each property may have shrunk (i.e., $X_k^F \subseteq X_k^I$, where X_k^F is the final value set), and (2) each constraint has been assigned a status (e.g., violated constraints are assigned the "unsatisfied" status).

In order to eliminate infeasible property values, the constraint manager sends *evaluation requests* to existing constraint-based systems or CAD tools capable of evaluating constraints. Each request is composed of a *subnetwork* (*G* in algorithm), i.e., a subset of the constraint network C_{i+1}, including constraints, properties, and their current value sets. In order to leverage existing constraint-based systems and CAD tools, they are *encapsulated*, i.e., a software wrapper is built around each of them providing a standard interface (see Fig. 4). Each encapsulated system is referred to by the constraint manager as a *constraint evaluator* (*E* in algorithm). This encapsulation approach does not require modification of legacy code, allows constraint propagation to be distributed, and simplifies the constraint manager engine.

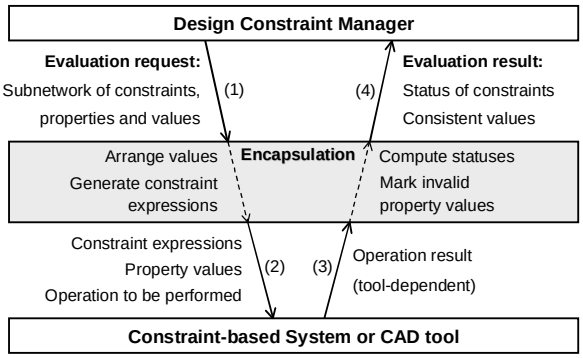

Fig. 4. Evaluation of a group of constraints.

Each request includes the first constraint in the queue *Q* and all other constraints in *Q* which the selected evaluator *E* can evaluate. When a wrapper receives a request, which includes the evaluation function identifiers for each constraint, it generates input constraint expressions as required by the constraint evaluator, and communicates them along with the property information to the evaluator[1]. For example, for the "totalCost" function identifier, an expression of the form $\sum_i cost_i = Cost_{total}$ is generated. Using the output result from the evaluator, the wrapper builds a response message and sends it to the constraint manager. This message contains the statuses of the constraints and the new value sets for each constraint argument after infeasible value elimination[2].

The status of each constraint c_j after the algorithm has finished depends on the value sets of c_j's arguments and belongs to one of the following:

- *satisfied*, if c_j is not violated, i.e., there can be *no conflict*, as long as the values of c_j's arguments belong to their initial value sets;
- *unsatisfied*, if c_j is violated, i.e., there is a *conflict*, because at least one of the final value sets is empty; or
- *consistent*, if c_j *may be* violated in the future, i.e., there is a *possible conflict*, which will happen if a combination of values from the initial value sets is chosen that violates the constraint.

[1] For some tools, constraint expression generation is not required.
[2] If the CAD tool accepts just one value per variable, steps (2) and (3) in Fig. 4 may need to be executed more than once for a single request.

2.6. Constraint violation handling

The history H_{i+1} contains the necessary information for the notification manager to handle constraint violations. In order to learn about newly detected conflicts, the notification manager examines H_{i+1} to find the status of each constraint in C_{i+1} after constraint propagation. Since it is critical to include history information in the notification, this manager examines H_{i+1} to find the design steps that led to each conflicting value. Finally, since it is necessary to find who should be notified of each conflict, this manager examines H_{i+1} to find which designers are associated with each property value involved in a violated constraint.

We address the problem of controlling the notification policy by introducing the concept of *history query functions*, i.e., database query expressions [12] applied to H_{i+1} by the notification manager. These functions are used to select from H_{i+1} which designers must be notified of a given type of conflict, for example:

```
select Designer
where InterestedIn(Designer, Property) and
      InvolvedIn(Property, PowerRequirement) and
      Violated(PowerRequirement)
```

In this example, the notification manager selects from H_{i+1} all designers "interested" in the properties "involved" in a constraint of type "PowerRequirement". Similarly, these functions are used to select the information that must be included in the notification (e.g., the CAD tool executions that led to the conflicting property values). We have extended DDDL to describe these query functions and thereby provide the ability for team members to control the notification policy. For reasons of space we forego a discussion of this extension in this paper.

3. CONSTRAINT MANAGEMENT IN MINERVA II

We have implemented the CCM methodology by adding constraint generation, propagation, and violation handling capabilities to the Minerva II design process manager [15]. As illustrated in Fig. 3, Minerva II has been modified by incorporating a Design Constraint Manager (DCM) module and a Notification Manager (DENIM) module. The Constraint Manager is an entirely separate program that could be used with other design process managers, provided that they followed the standardized protocol expected by the Constraint Manager. DENIM is configured with the DDDL description maintained by Minerva II.

Designers interact with Minerva II to formulate and solve design problems by following the *problem solving cycle* (Fig. 5). Designers may select any problem that is ready to be solved at any time, and then solve it with Minerva II's support. If the problem seems too complex to be addressed directly, it may be decomposed into simpler problems (Decomposition). Otherwise, designers can ask Minerva II to generate a sequence of CAD tools (called a plan) suitable for solving the problem. Minerva II consults available frameworks and returns all possible sequences (Plan Generation). Designers choose one or more of these sequences and Minerva II executes the chosen sequences and returns a result. If the result is satisfactory, the problem is marked as solved. Otherwise, designers

may backtrack or address another problem (this last option is not shown for simplicity). The coordination process is inserted at the points in the problem solving cycle that correspond to design steps, and is executed in cooperation by Minerva II, the Constraint Manager, and DENIM.

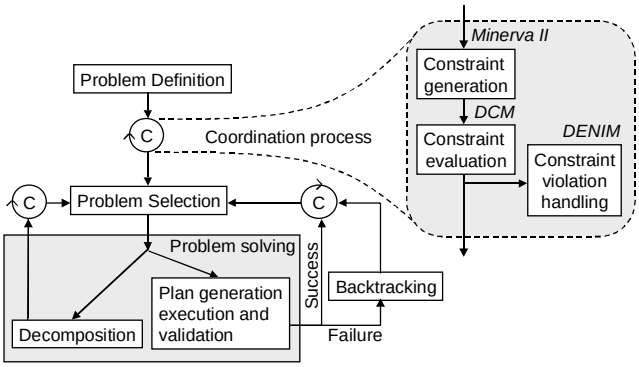

Fig. 5. Design problem solving cycle in improved Minerva II.

To implement the CCM methodology, we have encapsulated a set of tools, frameworks, and constraint-based systems to design integrated microelectromechanical systems (MEMS) with Minerva II, including analog design tools (Spice, Koan/Anagram [4], and Saber), MEMS CAD tools (Intellicad, Mistic [17]), digital design tools (by Cadence and Design Acceleration), and the constraint propagation library *ConstrLib* [5], which operates on most types of arithmetic and logical constraints, where variables can take both discrete and continuous values.

3.1. A conflict management session with Minerva II

Since a key goal of constraint management is to guide smart redesign trade-offs, an essential element of the proposed methodology is the user interface with which designers interact to manage conflicts. We describe this interface on the basis of an illustrative example.

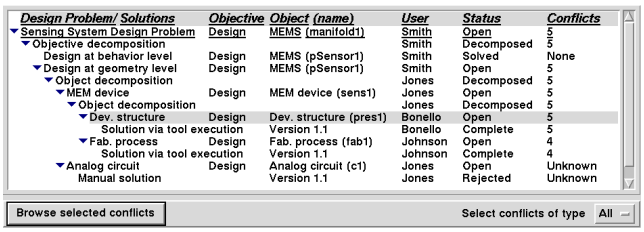

Fig. 6. Design problem status window in Minerva II.

Consider the team-based design of a MEMS system including pressure microsensors and analog circuitry. Examples of top-level constraints such a system must meet are timing, yield, resolution, dynamic range, and cost. Fig. 6 shows a snapshot of the Minerva II design problem status window at a point in the design process. Designers may search for areas of the design that are likely to require redesign by examining the right most column (Conflicts), which shows the number of conflicts affecting the properties associated with each design problem. Specific types of conflicts can be viewed, e.g., only timing constraints (see lower right button). In Fig. 6, the "Dev. Structure" problem (selected row in white pane), for example, has 5 constraint violations.

In our example, the estimated yield has fallen below its minimum required value. Fig. 7 shows the window by which DENIM notifies the sensor device designer of this constraint violation. The figure shows that the device cross section and the process module specification are involved in the conflict. The notification suggests possible resolution strategies, in particular to reduce the top-layer thickness in the cross section[1].

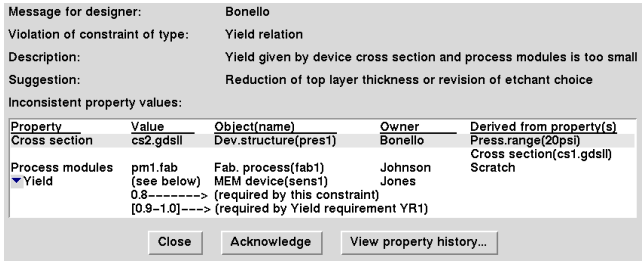

Fig. 7. Constraint violation notification. The history of any of the conflicting property values may be examined by pressing the "View property history" button. The violated constraint was evaluated using a process compilation CAD tool [17] to compute the yield, resulting in a value of 0.8. This value was not consistent with the yield requirement, which required yield to be at least 0.9. Thus, there is a conflict as yield has no feasible value.

A crucial element of searching for an effective design fix is to review the history behind the decisions that cause the conflicts. DENIM makes such a review possible. Fig. 8 illustrates doing so for one of the properties involved in the yield conflict, the cross section. The history indicates that the objective of the design problem whose solution set the cross section value was to optimize the dynamic range of the sensor. The goal was to achieve at least 20 psi, and the achieved result was 25 psi[2]. This 5 psi margin suggests that it might be possible to fix the yield conflict by reducing the top-layer thickness in the cross section, without violating the dynamic range requirement.

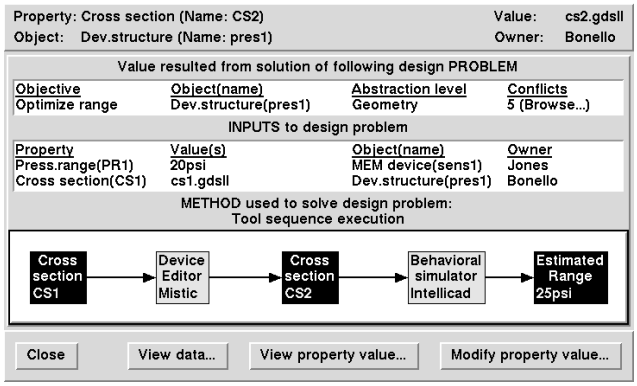

Fig. 8. Property history browsing. The formulation and solution of the design problem leading to the value of the cross section are shown.

Before making such a change, the designer may want to explore other effects of changing the cross section, possibly in other design domains. The constraint and property browser shown in Fig. 9 supports this exploration by providing the ability to dynamically

[1] Conflict resolution suggestions can be captured in our DDDL extension.
[2] Note that designers do not need to pursue time-consuming manual input of this design history - Minerva II captured it automatically as designers designed with its support.

expand at request portions of the constraint network graph involving properties of any given design object. In this example, expanding the yield portion shows first that the yield depends on the cross section (via constraint "Yield relation", i.e., the relation by which yield is calculated from the process modules and the cross section). Further expansion (left to right in snapshot) shows that "resolution", too, depends on the cross section. Not visible within the pane is how the dynamic range also depends on the cross section. Fig. 9 also shows both the "Yield requirement" and the "resolution requirement" as violated. The fact that both constraints are violated encourages the device designer to reduce the top-layer thickness in the cross section, which will affect both constraints (as well as the dynamic range). The effect of this change on the entire design can be viewed after constraint propagation. In this case, both yield and resolution are favorably affected by the change. Thus, in this example, a tie has been made between product design aspects (dynamic range and resolution) and a process aspect (yield) when that tie could enable smart redesign. Although not shown, we can imagine a process engineer using the same browser to analyze other options to solve the yield conflict.

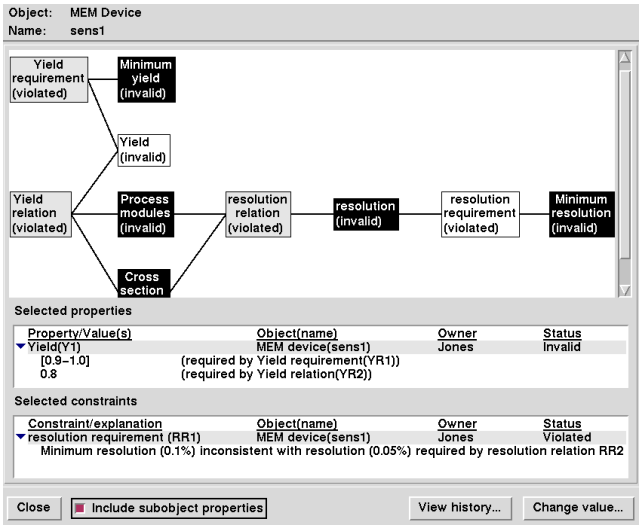

Fig. 9. Constraint and property browser. In the graph, property nodes are dark, constraint nodes are light-shaded, and selected nodes are white. The user is currently examining the information generated by the constraint manager about the "Yield" property and the "resolution requirement" constraint.

3.2. Advantages and applicability

The CCM methodology detects complex conflicts early in the design process, and provides critical information to analyze the underlying reasons and to discover the best solution trade-offs. All interactions among the various aspects of the design are considered together throughout the design process, without overwhelming designers with information.

Implementing this methodology requires developing wrappers for CAD tools and constraint-based systems for the use of the constraint manager. To reduce this implementation effort, we have developed reusable wrapper "templates" including capabilities required by most wrappers, such as mechanisms to communicate with the constraint manager. However, the evaluation of some constraints may not be amenable to automation. Thus, some constraints may have to be managed the conventional way, i.e., by running tools and visually comparing results with specifications. In this case, Minerva II can still manage the information related to these constraints. Finally, although constraint propagation is distributed in CCM, it is computationally expensive. Fortunately, however, it can be run during off-peak computation hours, e.g., at night.

4. CONCLUSIONS

Design teams are becoming larger and more multidisciplinary. A single designer or CAD tool can no longer account for the complexity of the interactions among the numerous aspects of a design. The CCM methodology represents these interactions as constraints, applies constraint propagation to detect conflicting decisions, guides the attention of designers toward the most conflicting areas, and helps designers identify promising resolution strategies. In doing so, CCM takes advantage of existing capabilities in design process management, constraint propagation, and CAD tool technology, and enhances these capabilities to make early conflict detection and resolution feasible. Work is currently underway to (1) improve performance by making the constraint manager "reuse" previous constraint evaluations, and (2) notify designers of events other than constraint violations (e.g., when a new constraint on cost has been introduced by a manager of the design project).

5. ACKNOWLEDGEMENTS

The authors are grateful to the reviewers, to Anne Gattiker, and to Jingyan Zuo for their helpful comments. This work has been funded in part by a scholarship from Spain's Science and Education Department.

6. REFERENCES

[1] C. Bessiere and J. Regin, "Arc consistency for general constraint networks: preliminary results", *Proc. IJCAI'97*: 398-404.

[2] F.L. Chan, M.D. Spiller, and A.R. Newton, "Weld - an environment for web-based electronic design", *Proc. 35th DAC*, June 1998.

[3] T.F. Chiueh and R.H. Katz, "Intelligent VLSI Design Object Management", in *Proc. EDAC*, pp. 410-414, 1992.

[4] J. Cohn et al., "KOAN/ANAGRAM II: New tools for device-level analog placement and routing", *IEEE JSSC*, 26(3):330-342, March 1991.

[5] J. D'Ambrosio, *ConstrLib: An Interval Constraint Propagation Library*, AI Lab, The University of Michigan, 1998.

[6] M. Fitting, *First-Order Logic and Automated Theorem Proving*, Springer Verlag, New York, second edition, 1996.

[7] S.M. Fohn et al., "A Constraint-system Shell to Support Concurrent Engineering Approaches to Design", *AI in Engineering*, (9):1–17, 1994.

[8] S.T. Frezza, S.P. Levitan, and P.C. Chrysanthis, "Requirements-based Design Evaluation", *Proc. 32nd DAC*, June 1995.

[9] D. Kuokka et al., "A parametric design assistant for concurrent engineering", *AI-EDAM*, no. 9: 135-144, 1995.

[10] M.Jacome and S.Director, "A formal basis for design process planning and management", *IEEE Trans. CAD*, 15(10):1197–1211, Oct. 1996.

[11] V. Kumar, "Algorithms for Constraint Satisfaction", *AI Magazine*, 13(1):32–44, 1992.

[12] A. Silberschatz et al., *Database System Concepts*, McGraw-Hill, 1996.

[13] P. R. Sutton, J.B. Brockman, and S.W. Director, "Design Management Using Dynamically Defined Flows", *Proc. DAC*: 648–653, 1993.

[14] P. R. Sutton and S. W. Director, "A Description Language for Design Process Management", *Proc. 33rd DAC*, 1996.

[15] P. R. Sutton and S. W. Director, "Framework Encapsulations: A New Approach to CAD Tool Interoperability", *Proc. 35th DAC*, June 1998.

[16] K.O. ten Bosch et al., "Design Flow Management in the Nelsis CAD Framework", *Proc. 28th DAC*: 711–716, June 1991.

[17] M. Zaman, *MISTIC User's Guide*, Univ. of Michigan, 1997.