Feature-Based Design Evaluation through Constraint Structure Unification

Kurt Godden
Computer Science Department
General Motors Research Laboratories
30500 Mound Rd, P.O. Box 9055
Warren, MI 48090-9055
Internet: kgodden@cmsa.gmr.com

Abstract

Constraint Structures are presented as a new knowledge structure to encode both engineering part designs and constraints on those designs. Using a common representational form for both designs and constraints results in a number of practical advantages for a design evaluation system grounded on constraint structures. First, a graph unification algorithm generalized to operate on constraint structures is employed as a mathematically elegant means of applying constraints to designs. As a side-effect of constraint structure unification, constraints are attached to design representations. Once the constraints become a part of the design representation, a part is able automatically to re-evaluate relevant constraints following interactive changes to design parameters without re-invoking the unification algorithm.

Overview

In the production of engineering parts, a proposed design must be evaluated against a variety of sometimes conflicting constraints. In practice such evaluation is often performed by hand using a paper-based manual that lists the constraints, or perhaps by an individual or group of individuals relying upon their experience in the domain. Such non-automated analyses suffer from several problems. For example, if the design of the part changes downstream in the production process, it must be re-submitted for evaluation. This cycle may iterate and result in lengthy delays. Further, any manual-based evaluation is naturally subject to human error.

As manufacturing companies increasingly rely on automated design tools, it becomes more and more feasible to construct automated design analysis tools to help automate the post-design evaluation process. Such tools may reduce the overall production time as well as the degree of error since a properly designed automated tool would not forget to re-examine a previously tested constraint following a minor design change that may affect that constraint.

Automated constraint processing constitutes an active field of research in the Artificial Intelligence community. Under constraint satisfaction paradigms, satisfactory designs are those where all constraints relevant to the design are satisfied [1, 2, 3, 4]. However, in a production manufacturing environment it is often necessary to accept designs that deliberately violate certain constraints. For example, in engineering design it may be desirable to violate one or more constraints in order to achieve an aesthetic product.

This paper presents a different paradigm for constraint processing. The goal of this paradigm is to facilitate the construction of automated post-design analysis tools that 1) employ active feature-based descriptions of designs, and 2) support constraint-based design evaluation that does not necessarily require the satisfaction of all relevant constraints. These goals are accomplished by means of a new knowledge structure called a constraint structure which may be used to represent both part designs as well as constraints on those designs. This common representation enables a generalization of a graph unification algorithm to apply constraints to part designs. As a side-effect of constraint structure unification, constraints are attached to design representations, becoming integral components of the design. Once the constraints are attached, a part is able automatically to re-evaluate relevant constraints following interactive changes to design parameters without re-invoking the unification algorithm.

The remainder of this paper describes the operation of an implemented proof-of-concept research software prototype for feature-based design evaluation through constraint structure unification. The examples involve a relatively informal discussion around hypothetical widgets and constraints in order to focus attention on the principles of constraint structure unification. However, the actual domain of development for the software prototype involves real-world engineering parts and
constraints relevant to their fabrication. Furthermore, this software system is not ad hoc in nature, being grounded in a mathematically formal theory of constraint structure unification developed by the author.

The system operates by iteratively applying each constraint from a database of constraints to a part design and determining for each constraint if it is satisfied, violated, contingent, inapplicable or irrelevant. Each of these possibilities will be discussed below. The system advises the user when a proposed design violates a constraint or lacks design parameters needed to evaluate a constraint. In both of these cases, the user may (or may not) immediately modify one or more design parameter values to satisfy the constraint in question. The constraints, once tested, are attached to the design representation, allowing the constraints to automatically re-check themselves if they are affected by interactive design changes.

**Constraint structures and unification**

Part designs as well as constraints may be represented by constraint structures, which constitute a generalization of feature structures. Informally, a feature structure [5] is a recursive data structure comprised of a set of symbolic feature-value pairs. Feature values may be either atomic or complex, a complex value being another feature structure. Atomic feature values are either constants or variables, where a variable is represented by an empty feature structure. If we regard constant values as name constants of a formal first-order lambda calculus, then we may generalize feature structures into constraint structures by allowing the "atomic" values to be additional meaningful expressions of the lambda calculus. In particular, part designs are represented by constraint structures that have only name constants as atomic values; whereas constraints on part designs are constraint structures that also permit lambda-expressions as atomic values. (By the term "name constant" I mean to indicate the concept as used in formal language theory. Thus I refer not only to symbolic names such as "galvanized-steel", but also to numeric names such as "25" and any other data type defined as atomic.) These lambda-expressions represent the content of some constraint that is to apply to one or more name constants of a part design in positions that correspond structurally to those lambda-expressions.

For example, if we represent a widget as:

```plaintext
widget =
    { < name , willy's-widget >
      < feature-x , 43 >
      < feature-y , 22 >
    }
```
we could represent the constraint stating that feature-x should be at least as large as feature-y by the constraint structure:

```plaintext
xy-constraint =
    { < feature-x , λx [ x ≥ y ] >
      < feature-y , λy [ x ≥ y ] >
    }
```

In order to apply the xy-constraint to the widget, we employ a variation of the graph unification, or feature-structure unification algorithm [5]. We refer to this algorithm as *constraint structure unification*, or cs-unification. Informally, cs-unification produces a unifier from two constraint structures such that if `< f , v >` is a feature-value pair in the unifier then either (1) feature `f` appears in one of the two unificands but not in the other; or (2) `< f , v1 >` appears in one unificand, `< f , v2 >` appears in the other unificand, and `v` is the cs-unification of `v1` and `v2`. Two identical name constants cs-unify to produce the same constant. If a lambda-expression is cs-unified with a constant the unifier may (somewhat inaccurately) be conceived as the result of their β-reduction. (β-reduction is also known as lambda-application or just functional application. For example, the β-reduction of `λx [ x > 2 ] 3` is `[ 3 > 2 ]`, where the argument `3` simply replaces all bound occurrences of the formal parameter `x` in the functor `λx [ x > 2 ]`.) The constraint structure resulting from the cs-unification of the input constraint structures is an information preserving combination of both unificands. In effect, the constraint is attached to the representation of the part. The cs-unification of the above two constraint structures yields:

```plaintext
xy-constraint-and-widget =
    { < name , willy's-widget >
      < feature-x , [ 43 ≥ y ] >
      < feature-y , [ x ≥ 22 ] >
    }
```

More accurately, modifications to binding environments are performed instead of applying β-reduction. Hence, the unifier is more precisely:

```plaintext
xy-constraint-and-widget =
    { < name , willy's-widget >
      < feature-x , [ x ≥ y ] >
      < feature-y , [ x ≥ 22 ] >
    }
```

where `x ← 43` is maintained in a binding environment for the first occurrence of `[ x ≥ y ]`, and `y ← 22` for the second.

Notice that once such a constraint has been applied and attached to a part representation, the constant value (e.g. 43) that had appeared in the original part representation is replaced by a formula. However, the original constant value remains accessible through that formula’s associated binding environment. This is important because it allows for repeated application of additional constraints onto the same structure. For example, the following constraint

```plaintext
feature-x-constraint =
    { < name , willy's-widget >
      < feature-x , λx [ x ≥ y ] >
    }
```

could be applied to the xy-constraint-and-widget structure above to produce the unifier

```plaintext
xy-constraint-and-widget =
    { < name , willy's-widget >
      < feature-x , λx [ x ≥ y ] >
      < feature-y , λy [ x ≥ y ] >
    }
```

In order to apply the xy-constraint to the widget, we employ a variation of the graph unification, or feature-structure unification algorithm [5]. We refer to this algorithm as *constraint structure unification*, or cs-unification. Informally, cs-unification produces a unifier from two constraint structures such that if `< f , v >` is a feature-value pair in the unifier then either (1) feature `f` appears in one of the two unificands but not in the other; or (2) `< f , v1 >` appears in one unificand, `< f , v2 >` appears in the other unificand, and `v` is the cs-unification of `v1` and `v2`. Two identical name constants cs-unify to produce the same constant. If a lambda-expression is cs-unified with a constant the unifier may (somewhat inaccurately) be conceived as the result of their β-reduction. (β-reduction is also known as lambda-application or just functional application. For example, the β-reduction of `λx [ x > 2 ] 3` is `[ 3 > 2 ]`, where the argument `3` simply replaces all bound occurrences of the formal parameter `x` in the functor `λx [ x > 2 ]`.) The constraint structure resulting from the cs-unification of the input constraint structures is an information preserving combination of both unificands. In effect, the constraint is attached to the representation of the part. The cs-unification of the above two constraint structures yields:

```plaintext
xy-constraint-and-widget =
    { < name , willy's-widget >
      < feature-x , [ 43 ≥ y ] >
      < feature-y , [ x ≥ 22 ] >
    }
```

More accurately, modifications to binding environments are performed instead of applying β-reduction. Hence, the unifier is more precisely:

```plaintext
xy-constraint-and-widget =
    { < name , willy's-widget >
      < feature-x , [ x ≥ y ] >
      < feature-y , [ x ≥ 22 ] >
    }
```

where `x ← 43` is maintained in a binding environment for the first occurrence of `[ x ≥ y ]`, and `y ← 22` for the second.

Notice that once such a constraint has been applied and attached to a part representation, the constant value (e.g. 43) that had appeared in the original part representation is replaced by a formula. However, the original constant value remains accessible through that formula’s associated binding environment. This is important because it allows for repeated application of additional constraints onto the same structure. For example, the following constraint

```plaintext
feature-x-constraint =
    { < name , willy's-widget >
      < feature-x , λx [ x ≥ y ] >
    }
```

could be applied to the xy-constraint-and-widget structure above to produce the unifier
two-constraints-and-widget =
{ <name, willy's-widget>
  <feature-x, \{[x \leq y] \wedge [x2 \leq 35]\}>
  <feature-y, \{[x \leq y]\}>
},
where x , x2 \leftarrow 43 in \{[x \leq y] \wedge [x2 \leq 35]\}, and
y \leftarrow 22 in \{x \leq y\}.

In order to obtain this result, the cs-unification
algorithm has to be able to unify the formula \{x \leq y\}
with the lambda-expression \lambda x2 \{x2 \leq 35\}. This is
accomplished by retrieving the binding of the variable x
in the former from its associated binding environment,
then using that value to bind the variable x2 in the
lambda-expression, and finally conjoining the latter's
embedded formula with the first formula.

Note that while these examples of constraint formulae
involve simple numeric relational operators, there is no
arbitrary restriction on the complexity of constraints that
may be represented. The full expressive power of a first-
order predicate calculus with the lambda-operator is
available. In addition, the primitive relations of this
calculus may be any well-defined n-ary relations. In
practice, a developer of a system based on cs-unification
can essentially use any boolean expression of the
programming language of implementation as the language
in which to express constraints. Common LISP
expressions are used to represent constraints in the
prototype being described. Newly-defined logical
connectives replace the LISP boolean connectives in
complex expressions in order to attain the desired
semantics as described in the next section.

**Constraint evaluation**

Note that the success of cs-unification does not depend
upon the truth value of any logical constraint formula.
Following the operation of cs-unification, the system may
test the status of a constraint proper by evaluating its
formula from the unified constraint structure in its full
binding environment. The full binding environment is
obtained from the constraint structure by collecting all the
individual binding environments associated with individual
occurrences of formulae. For example in the above unifier
xy-constraint-and-widget, it will be remembered that there
are two binding environments, one for each occurrence of
the formula \{x \leq y\}. One environment binds x and the
other binds y. The formula cannot be evaluated until both
environments are combined.

A constraint is satisfied if the formula is true in that
combined environment. This is the status of the
constraint in the example involving xy-constraint-and-
widget because x's value of 43 is greater than y's value, 22.
A constraint is violated if the formula is false in the
environment. This is the case with the second conjunct of
\{[x \leq y] \wedge [x2 \leq 35]\} in two-constraints-and-
widget.

In the event of such a constraint violation, the system
enters into a dialogue with the user allowing changes to
any of the variables in the combined environment.
Currently, it is the user's responsibility to correct (if
desired) that value or values giving rise to the violation.
Certainly in violations involving simple constraints such
as the ones explained here, the system could be modified
to recognize the offending parameter(s). However, in
general such recognition could prove to be very difficult
for complex constraint formulae (see example under
Further Research); hence, this form of intelligence is
left to the user in the current implementation.

If the user chooses to modify one or more variable
values, then all local binding environments referring to
the changed variable(s) must be updated for the new
value(s). Such propagation of new values can be
performed automatically, however. All relevant binding
environments are accessible from the node of a changed
variable because the affected constraints (implemented as
active objects with their binding environments) were
attached to that node during previous cs-unifications. As
this propagation is performed, each affected constraint
automatically re-evaluates itself since its satisfaction
status may change due to the newly received variable
value(s). Of course, any new violation will trigger
another dialogue asking the user for additional changes.
The current implementation performs no consistency
checking, relying on the user to notice and rectify
constraint conflicts.

A constraint's status may be a value other than
satisfied or violated. A third value is contingent. A
contingent constraint is represented by a formula whose
truth value cannot be determined because one or more of
its variables has no binding in the combined environment.
This condition is indicative of an incomplete part design
representation, as the following example will illustrate.
Suppose that incomplete-widget were to be tested against
the xy-constraint from above:

incomplete-widget =
  \{ < feature-x, 30 > \}
This cs-unification would result in:
contingent-xy-constraint-and-widget =
  \{ < feature-x, \{x \leq y\} >
  < feature-y, \lambda y (x \leq y) > \},
where x \leftarrow 30 is maintained in a binding environment for
\{x \leq y\}, and no binding is obtained for y.

Since there is no binding for y, the system assigns the
formula \{x \leq y\} an evaluation status of contingent.
That is, the formula's truth value is contingent upon
assigning a value to variable y. Of course, there is no
value for y because the original design specification of the
part in question is incomplete. Contingent constraints,
lke violated constraints, cause the system to enter a
dialogue with the user in which a value for the missing
design parameter(s) is requested. If the user chooses to
supply all such missing values, then the affected constraints evaluate themselves to determine their truth value.

A fourth constraint status is referred to as inapplicable, not to be confused with irrelevant constraints discussed next. This status was invented for pragmatic reasons, due largely to the fact that the system allows interactive design modifications for violated or contingent constraints as just described. Essentially, an inapplicable constraint is one in which the first-order formula is an implication whose antecedent happens to be false. As an example, let us use the constraint represented by the constraint structure

second-xy-constraint =
\{ < feature-x, λx \[x < 30\] \rightarrow [y < 20] > \\
< feature-y, λy \[x < 30\] \rightarrow [y < 20] > \}

and the part represented by the first widget, repeated here,

gadget =
\{ < name, willy's-widget > \\
< feature-x, 43 > \\
< feature-y, 22 > \}

which cs-unifies with second-xy-constraint to produce

second-xy-constraint-and-widget =
\{ < name, willy's-widget > \\
< feature-x, [[x < 30] \rightarrow [y < 20]] > \\
< feature-y, [[x < 30] \rightarrow [y < 20]] > \},

where x \rightarrow 43 and y \rightarrow 22 as above.

When the system evaluates this result it determines that the constraint's antecedent is false, and therefore classifies the status of the constraint as inapplicable. An inapplicable constraint causes nothing special to occur until after all constraints have been processed by the system, at which time a summary of all attached constraints is presented with each constraint's status. It would seem inappropriate, if not actually annoying, for a user to be presented in this summary with a listing for a constraint whose pre-conditions (in the form of a false antecedent in an implication) are not met. Therefore, the system suppresses from this post-processing summary any display of inapplicable constraints. It might appear simpler just to discard inapplicable constraints during cs-unification so that they are not attached to the part design in the first place. This approach can lead to difficulties, however, as the following scenario will illustrate.

Suppose that the second-xy-constraint were applied to the widget before any other constraints, found to be inapplicable and discarded by the system. Suppose further that following this action, the first xy-constraint and the feature-x-constraint were applied. We have already seen how the feature-x-constraint (which states that the value associated with feature-x should be no more than 35) is violated by this part and the user is offered an opportunity to change the value associated with feature-x.

Suppose that the user decides to change this value from 43 to 25. This particular value would satisfy both the feature-x-constraint since it is less than 35, and the the xy-constraint since 25 is still greater than 22, the value of y. However given this new value for feature-x, the constraint expressed by the second-xy-constraint would no longer be inapplicable. In fact it would be violated due to the value of y being greater than 20.

It can now be seen that if the second-xy-constraint were discarded, then the violation could no longer be determined. It was for this reason that the evaluation status of inapplicable was created to allow the appropriate suppression of summary messages, while retaining the constraint formula in question for re-evaluation in the event of a design change. One can think of an inapplicable constraint as being dormant until it is possibly awakened by a design change that satisfies its preconditions.

Finally, while \([P \rightarrow Q]\) and \([\neg P \lor Q]\) are logically equivalent, if \(P\) is false, then the former would have a status of inapplicable and the latter satisfied. The truth value of both, of course, would be true. While binary truth value is a logical concept, the four-valued status of a formula is a pragmatic concept. The disjunction \([P \lor Q]\) can be inapplicable if and only if either \(P\) or \(Q\) is inapplicable, and neither \(P\) nor \(Q\) is satisfied or contingent. The author has developed a recursive, albeit somewhat lengthy definition that defines the status of all complex formulas based upon the status of their constituents.

Irrelevant constraints

There are cases where a constraint is truly irrelevant and can be discarded. This situation is characterized by the failure of two input constraint structures to cs-unify. Consider the attempt to cs-unify the feature-x-constraint, repeated here,

feature-x-constraint =
\{ < name, willy's-widget > \\
< feature-x, λx2 \[x < 35 \] > \}

with the part
wonka-widget =
\{ < name, wonka's-widget > \\
< feature-x, 40 > \}.

Recall that two constraint structures must cs-unify along corresponding paths. Thus, the algorithm will compare the two constants willy's-widget and wonka's-widget, which are the corresponding values along the path name. Recall that cs-unification is a pattern matching algorithm, and constants only match if they are identical. Therefore, cs-unification fails in this example, and the feature-x-constraint is not attached to the representation for wonka-widget; it is discarded.
Further research

It has been mentioned already that the current system relies on the user's intelligence in two areas. The first involves recognition of which design parameters, represented by variables in constraint formulae, are crucially involved in a constraint violation where the constraint formula in question has multiple variables. For example, if \([x = 25] \land [y = 30]\) were known to be false where \(x + 25\) and \(y + 35\), then clearly a prudent system would ask the user only to correct \(y\). However, if the formula were \([x < y] \land [y < x2]\), where \(x + 25\), \(y + 35\), and \(x2 + 30\), then it is not at all clear if \(y\) or \(x2\) (or both) is the culprit and requires a new value. Clearly, this is one area for interesting additional research.

The second area previously mentioned that is in need of additional work involves consistency of sets of constraints. Mackworth [3] provides algorithms to transform constraint networks into consistent nets, and similar algorithms are called for in the case of constraint structures.

A third area being actively pursued involves the notion of quantification. As described to this point, constraint structure unification requires that the first-order lambda calculus used to express the content of a constraint in a constraint structure be a quantifier-free lambda calculus. However, without quantifiers it would be impossible to express the constraint, with complex widgets that have multiple occurrences of both feature-\(x\) and feature-\(y\) for example, that there must be at least one value of feature-\(x\) that is greater than all values of feature-\(y\). Such a constraint could be represented by the constraint structure:

\[
\text{quantified-xy-constraint} = \begin{cases} < \text{feature-x}, \lambda x \exists y [x > y] > \\ < \text{feature-y}, \lambda y \exists x [y > x] > \end{cases}
\]

While the constraint expression \(\lambda x \exists x \forall y[x > y]\) may look unusual in that the variable \(x\) appears to be in the scope of both the lambda operator and the existential quantifier, it is nonetheless a syntactically well-formed expression. The solution is to iteratively apply this constraint structure to that of the design object being evaluated, restricting the domain of the quantified variables to the union of the individual local binding environments created during cs-unification. This solution has been implemented in the software and its theoretical formalization is underway.

Summary and conclusions

Constraint structures have been presented as a generic knowledge structure in which to represent both engineering designs and constraints on those designs. Representing both parts and constraints with the same knowledge structure allows us to use a modified unification algorithm as a means of applying a constraint to a part. All of the concepts and techniques described above have been implemented in a proof-of-concept software prototype for post-design engineering parts analysis.

Several advantages accrue by employing this methodology for constraint processing. One such advantage is that the design accumulates its relevant constraints as a side-effect of the cs-unification algorithm. Therefore, previously tested constraints may be re-evaluated by the design object immediately following an interactive change to a design parameter without re-invoking the unification algorithm.

Constraints that are irrelevant to a particular design object may be distinguished by cs-unification failure. Hence these constraints may be discarded by the system and not carried around as excess baggage.

Constraint attachment also facilitates a summary of all constraints applied to a part design. Only the constraints relevant for a part are included in this summary, and implementing constraints as active objects with their bindings and their evaluation status permits the system to suppress from this summary those inapplicable constraints whose preconditions remain unmet after all processing and design changes have been made.

Constraint evaluation is a separate operation from cs-unification. Thus it is not only the satisfied constraints that are attached to a part representation; but also violated, contingent and even inapplicable constraints are attached as well. We have seen how previously tested but inapplicable constraints remain relevant and may be re-activated by interactive design changes. Contingent constraints are indicative of incomplete design representations in a part and may thus be used to provide user feedback to this effect.

While cs-unification has been presented here for post-design analysis, it is clear that this new paradigm for constraint processing can be used in the design process itself. By employing a backtracking search algorithm and a unification algorithm that was sensitive to the status of the logical formulas, one could explore the space defined by those constraints relevant to an initial design and which have a status of satisfied. Design parameter changes would be made at choice points in this search space.

Finally, it should be stated that one of the motivating factors in the development of constraint processing via cs-unification was the desire to allow for deliberate violations of constraints in a real-world environment. In manufacturing companies, constraints often represent ideals which are desirable, but which may have to be violated in order to achieve a design that satisfies other, less quantitative, considerations. The implementation of cs-unification as presented here describes how a system can intelligently apply constraints to a design, informing
a user of problems that are encountered, while leaving final authority with the user, where this authority properly belongs, either to accept constraint violations or to modify the design.

References


