

The Designers' Workbench: Using Ontologies and Constraints for Configuration

David W. Fowler
University of Aberdeen
Aberdeen, UK

Derek Sleeman
University of Aberdeen
Aberdeen, UK

Gary Wills
University of Southampton
Southampton, UK

Terry Lyon
Rolls-Royce plc
Derby, UK

David Knott
Rolls-Royce plc
Derby, UK

Abstract

Typically, complex engineering artifacts are designed by teams who may not all be located in the same building or even city. Additionally, besides having to design a part of an artifact to be consistent with the specification, it must also be consistent with the company's design standards.

The Designers' Workbench supports designers by checking that their configurations satisfy both physical and organisational constraints. The system uses an ontology to describe the available elements in a configuration task. Configurations are composed of features, which can be geometric or nongeometric, physical or abstract. Designers can select a class of feature (e.g. Bolt) from the ontology, and add an instance of that class (e.g. a particular bolt) to their configuration. Properties of the instance can express the parameters of the feature (e.g. the size of the bolt), and also describe connections to other features (e.g. what parts the bolt is used to hold together).

1 Background

Engineering designers typically have to find configurations of parts that implement a particular function. The engineering design process is constraint orientated, and requires the recognition, formulation and satisfaction of constraints [LC02]. To assist designers, most organisations have built up a large number of design rules and standards, usually held as large volumes of text. Designers must try to ensure that their configurations satisfy these constraints, but it is often easy to overlook some. Novice designers often have a hard task in learning the constraints. On occasion, constraints may be modified or become obsolete. Also, new constraints may be added. In practice, it is often hard to find which constraints apply in a given situation. Additionally, in a collaborative environment, where many designers are working on subsections of a common component, it is common for changes made by one designer to affect the options available to another, but for this to go unnoticed until much later, causing expensive and time consuming redesigns.

It would clearly be useful to have some way of automating the design checking process, so that all applicable constraints are checked, without the designer having to manually initiate a search for the constraints. In our approach, we use an ontology to

describe the available features and check the constraints automatically. A constraint is specified by the conditions of applicability (which types of features it applies to), and the logical or mathematical expression that allows a constraint check to be performed.

2 Aims of Designers' Workbench

The Designers' Workbench deals mainly with problems that lie in the domain of configuration. Mittal and Frayman [MF89] define a configuration problem as selecting parts from an existing set of types, and connecting them using specified ports in such a way that certain constraints are satisfied and that particular functions can be performed by the resulting configuration. Brown [Bro98] discusses the adequacy of this definition. For our purposes, we will assume that ports can be described by properties of a feature, so that "normal" properties may have values that are simple data values to further describe the feature, whereas "port" properties have values that are other features. For an example, consider a class corresponding to bolts. An instance of this class will have properties that describe the size and shape of the bolt (real numbers or integers). There will also be properties that have values that are instances of other classes — for example, the `has_material` property might have a value `steel`, an instance of the `Material` class.

The current version of Designers' Workbench allows designers to build a configuration and to check that all the appropriate constraints hold. We have used the term features, rather than parts or components, as we wish to emphasise the fact that features can be abstract entities (such as temperatures, holes in other features, etc). Brown [Bro02] discusses a large number of different uses of the word "feature", including functional features that we are also interested in. The designer can select a feature class from an ontology, and add an instance of it to the configuration. Each instance can have properties that are defined for the class that it belongs to. The properties that are defined for a particular type of feature can be of two kinds: *datatype* (integers, strings, reals etc), which are parameters of the feature; and *objecttype* (other feature instances), which correspond to ports (connections to other features).

In a real engineering situation there may be many thousands of constraints, which means that it is easy to overlook some of them. We define constraints as generic, in that they apply to particular *types* of subconfigurations of features, rather than to specific features. It is not necessary to have any actual features defined before defining a constraint. For example, there may be a constraint that applies to neighbouring features such that if feature X is made of metal A, and feature Y is made of metal B, then the features are incompatible. This constraint could be added without any knowledge that such a pair of features exists in a design. Constraint checking becomes a process of finding such subconfigurations and checking that they satisfy the constraints. We have concentrated on checking that constraints are satisfied by a configuration produced by a human designer, rather than finding a solution. This has implications for tractability, in that solving a CSP is an NP-complete problem, whereas checking a solution can be done in polynomial time. The system has been implemented so that the human designer is free to use his or her engineering expertise to override constraints that are not deemed applicable to the current situation.

3 Related work

In this section we examine some approaches to the configuration problem, dividing them according to whether they use constraints, or ontologies, or both. General introductions to constraints can be found in [Kum92], [Smi95], or [Bar99], while introductions to ontologies are [CJB99], [McG02] and [SBF98].

3.1 Constraint-based approaches

Bowen et al. [BOS90] describe a constraint based language, LEO, which enables parts of a design to be represented as a collection of variables, with domains that are not necessarily finite (for example, rational numbers or reals). The system allows constraints to be specified over these variables. Constraint checking, rather than solving, is preferred, because:

A constraint monitoring system . . . allows the designer to exercise his creativity, while relieving him of the drudgery of making many routine inferences and checks, thereby ensuring that the design choices he makes are consistent with good performance in all significant aspects of the product's life cycle. Furthermore, constraint monitoring is less expensive, computationally, than constraint satisfaction. [BOS90]

A disadvantage of this approach, from the perspective of configuration, is that all variables must be declared at the outset. Also, the variables are not structured using classes. Variables corresponding to properties of features must be declared individually, whereas a structured system would associate properties with each feature class, and declare them automatically.

Mittal and Falkenhainer [MF90] introduce *Dynamic Constraint Satisfaction Problems* (DCSPs), for representing and solving configuration problems. A DCSP is similar to a CSP in that it consists of variables with prespecified finite domains, together with constraints that specify which values particular subsets of variables may take simultaneously. In addition, a DCSP variable may be *active* or *inactive*, and may switch during search. As well as standard constraints on variables, *activity constraints* may require that variables become active or inactive, dependent on the values (or activity) of other variables. For example, in a car configuration problem, variables `Package` and `Sunroof` might have domains `{luxury, deluxe, standard}` and `{sr1, sr2}` respectively. An activity constraint might require that if `Package=luxury` then `Sunroof` must be active (and can take a value from its domain). In this way, non-luxury cars will not have sunroofs, whereas luxury cars will have a choice of two sunroof types. "Normal" constraints (corresponding to CSP constraints) are only enforced on currently active variables. Algorithms are given to solve DCSPs. One disadvantage is that all variables must be specified in advance, making it awkward to represent problems whose solution may require several of the same type of component. For example, a configuration might require many bolts, but the exact number is not known before searching for a solution. Using a DCSP would require the declaration of a variable for every bolt that might possibly be needed.

Sabin and Freuder [SF96] define *composite CSPs* which allow configuration problems to be represented in a hierarchical fashion. In a composite CSP, some variables can be assigned a subproblem, rather than a simple value. In this way, components can be selected at a higher level, before being specified in terms of subassemblies. The

example used in [SF96] is also car configuration. The car is divided hierarchically: for example, the power plant system is divided into an engine, exhaust system, electrical system, and so on. Each of these can be represented by a variable that has a domain that is a subproblem. For instance, the engine variable could have a domain that has two values, gasoline engine and diesel engine. During search, if one of these values is selected, a subproblem will need to be solved, containing variables such as pistons, rods, valves, etc. An advantage of the approach is that existing CSP search algorithms can be easily adapted to solve composite CSPs, by dynamically adding and retracting variables and constraints as required.

3.2 Ontology-based approaches

Lin et al. [LFB96] give an ontology for describing products. The main decomposition is into *parts*, *features*, and *parameters*. Parts are defined as “a component of the artifact being designed”. Features are associated with parts, and can be either geometrical or functional (among others). Examples of geometrical features include holes, slots, channels, grooves, bosses, pads, etc. A functional feature describes the purpose of another feature or part. Parameters are properties of features or parts, for example: weight, colour, material. Classes of parts and features are organised into an inheritance hierarchy. Instances of parts and features are connected with properties *component_of*, *feature_of*, and *subfeature_of*.

McGuinness and Wright [MW98] describe the application of a description logic to configuration. *Concepts* can be defined, corresponding to the classes of an ontology, and *individuals* correspond to instances. The use of a description logic enables consistency checks to be made quickly. Forward chaining rules can be defined, which are “associated with concepts, but are applied only to individuals”. These rules are used to enforce constraints that are generic, i.e. defined over classes of objects, rather than over specific objects.

3.3 Combining constraints and ontologies

Stumptner et al. [SFH98] introduce an extension of CSP, *Generative CSP*, which uses complex variables, each of which has an associated type. The type of a variable determines its *attributes* (datatype properties), and its *ports* (objecttype properties). The types are formed into a simple hierarchy. The constraints here include activation constraints and compatibility constraints, introduced by Mittal and Falkenhainer [MF90]. In addition, resource constraints are used, being global constraints over all variables of a particular type.

Junker and Mailharro [JM03] describe a system, ILOG Configurator, that combines the power of description logic (to describe the parts used in a configuration), with constraint programming (to solve the configuration problem). The description logic uses classes that are either abstract or concrete. Concrete classes are the leaf classes of the ontology, corresponding to actual parts. Abstract classes are the nonleaf classes. Properties are used to describe the instances of a class. Alternative methods can be used to specify the instances that can be part of a solution, ranging from an explicit list of instances, to an implicit list, where instances can be freely chosen from an infinite universe of instances. Generic constraints can be defined in a constraint language that allows numeric and symbolic constraints. To solve a configuration problem, a description logic representation of the class hierarchy and the constraints are converted into a constraint satisfaction problem.

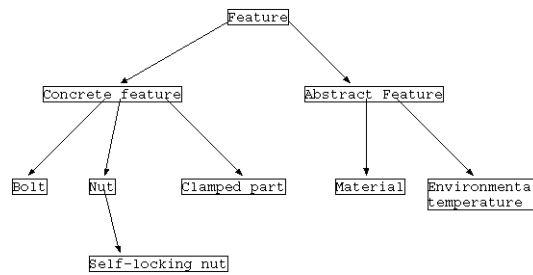


Figure 1: The class hierarchy of a simple configuration ontology

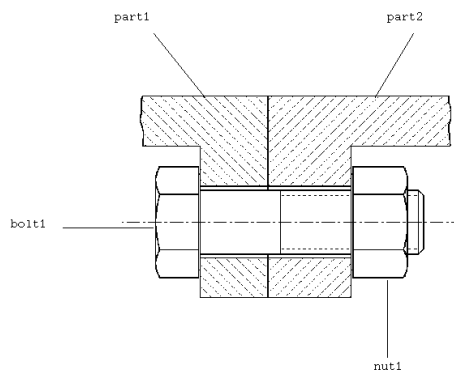


Figure 2: A bolted joint [image from French, Vierck and Foster, “Engineering Drawing and Graphics Technology”, McGraw-Hill Inc.]

Laburthe [Lab03] extends CSPs to cases where variables have domains that are taken from a hierarchy. This differs from the approach of the Designers’ Workbench (as well as that of [MW98] and [JM03] described above) in that we are concerned with constraints over values of properties of instances (ultimately datatype values). Laburthe’s approach aims to find the entities in a hierarchy that will satisfy the constraints. It is possible that this approach could be used to find suitable types for elements of a configuration.

4 An illustrative example

To illustrate the use of an ontology to describe a configuration, we will use the simple ontology whose class hierarchy is shown in Figure 1. We have used the concept of *feature* as the root of the ontology. Features have then been divided into *concrete features* (those that have a material), and *abstract features* (holes, temperatures, etc).

Using this ontology, we can describe the simple arrangement of a bolted joint shown in Figure 2, subject to a particular environmental temperature. This is shown in Figure 3. Example constraints might include:

- Any concrete feature must have a material with a higher operating temperature than the prevailing environmental temperature;

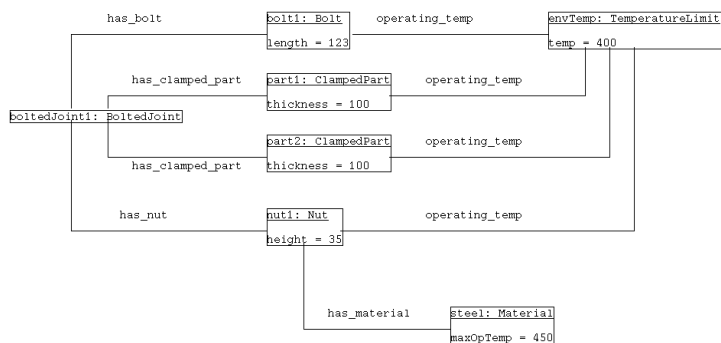


Figure 3: A configuration using the ontology

- The length of the bolt in a bolted joint must exceed the sum of the thicknesses of the clamped parts, plus the height of the nut. Note that for simplicity, we have ignored issues such as tolerances of dimensions, although this can be dealt with, for example by defining a Measurement class of feature, with a real valued properties dimension and tolerance.

The first constraint will apply to all features that have a `has_material` property and an `environmental_temperature` property defined. The second constraint is more complicated, and applies to all bolts, nuts, and clamped parts that are part of a bolted joint.

5 Functionality

An ontology written in DAML+OIL [DAM] describes available feature types. In the Designers' Workbench, the designer can select a feature class from the ontology and create an instance of that class. The property values of the instance can then be filled in: datatype values by literals of the appropriate type, and object type values by selecting an instance from a list of all instances of the appropriate type.

Part of the ontology (introducing the Bolt class) is shown below.

```
<daml:Class rdf:about="file:///D/DW/Onto2/DW_Onto4.daml#Bolt">
  <rdfs:label>Bolt</rdfs:label>
  <rdfs:comment><![CDATA[ ]]></rdfs:comment>
  <oiled:creationDate><![CDATA[2003-03-05T11:48:58Z]]></oiled:creationDate>
  <oiled:creator><![CDATA[dfowler]]></oiled:creator>
  <rdfs:subClassOf>
    <daml:Class rdf:about="file:///D/DW/Onto2/DW_Onto4.daml#Concrete Feature"/>
  </rdfs:subClassOf>
</daml:Class>
```

Constraints are handled in a two stage process:

- Identify feature values that should be constrained;
- Formulate a tuple(s) of values for each set of feature values, and check that the constraint is satisfied by these values.

The constraint processing uses RDQL (RDF Query Language) [JEN] to find the constrained features. A disadvantage of using RDQL is that it cannot return arbitrary numbers of values from a query, making constraints on arbitrary numbers of features impossible to express. For example, the constraint on the sum of thicknesses of clamped parts cannot be expressed. To get around this, it is possible to have separate constraints for 1, 2, 3, ..., n parts, where n is some arbitrary number, but this is clearly not ideal. Part of the future work will be to investigate other query languages or mechanisms to overcome this difficulty.

After using RDQL to extract the values that are constrained, SICStus Prolog [Swe03] is used for the process of checking that the constraints hold.

The RDQL query that locates features affected by the material temperature constraint is:

```
SELECT ?arg1,?arg2 WHERE
  (?feature,<dwOnto:has_material>,<?mat>),
  (?mat,<dwOnto:max_operating_temp>,<?arg1>),
  (?feature,<dwOnto:operating_temp>,<?optemp>),
  (?optemp,<dwOnto:temperature>,<?arg2>)
USING dwOnto FOR <namespace>
```

The values of the returned variables `?arg1` and `?arg2` are the maximum operating temperature of the material of the feature, and the current operating temperature, respectively. The check that the values must satisfy is represented by the SICStus predicate

```
op_temp_limit(MaterialMaxTemp, EnvironTemp) :-
  EnvironTemp =< MaterialMaxTemp.
```

Using the values of `?arg1` and `?arg2`, the query

```
op_temp_limit(MaterialMaxTemp, EnvironTemp).
```

is formed, and checked. This process is repeated for each set of values returned by the RDQL query, and for each constraint that has been specified.

The Designers' Workbench is written in the Java programming language. The OilEd tool [BHGS01] was used to create the ontology. The ontology is read using Jena [JEN], a Java library. Instances are created using Jena. The Jasper package [Swe03] is used to make calls to SICStus from Java. JGraph [JGR] is currently used for handling the graph-based display of the configuration.

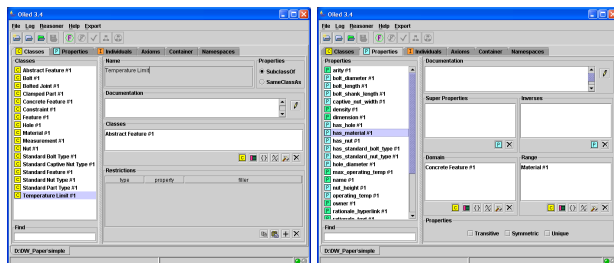


Figure 4: The OilEd ontology editor, showing classes (left) and properties (right)

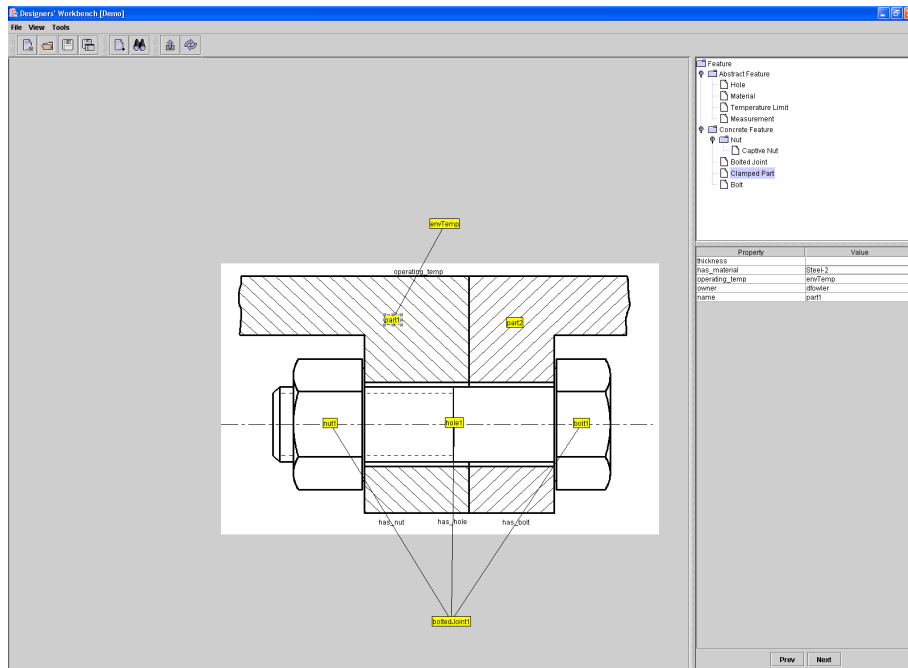


Figure 5: The Designers' Workbench

6 Additional features of the Designers' Workbench

A screenshot of the Designers' Workbench is shown in Figure 5, with closeups of the main panels in Figure 7. In this section, we describe the system's additional features.

6.1 Graph-based display of configuration

In the current implementation of Designers' Workbench, the designer can import a drawing and annotate it with features. The drawing is really a visual aid — the designer can “mark up” an existing drawing or construct a configuration without a drawing. Features can be selected from an ontology. Features that are added by the designer are shown as labels overlaying the background drawing. Properties that connect features are represented by arcs. Features can be selected, and their properties viewed and modified using the table displayed beneath the ontology. Datatype properties are set by typing values into the field, whereas object properties are set using a drop down list of values representing the valid possibilities for the property. For example, if the property `has_bolt` is specified to have range of class `Bolt`, the list will consist only of instances of `Bolt`.

6.2 Checking incomplete configurations

Before checking constraints, it is not necessary to specify values for every defined property of every feature. Instead, the designer can fill in values for whichever properties he or she desires, and request a constraint check. The RDQL query will only

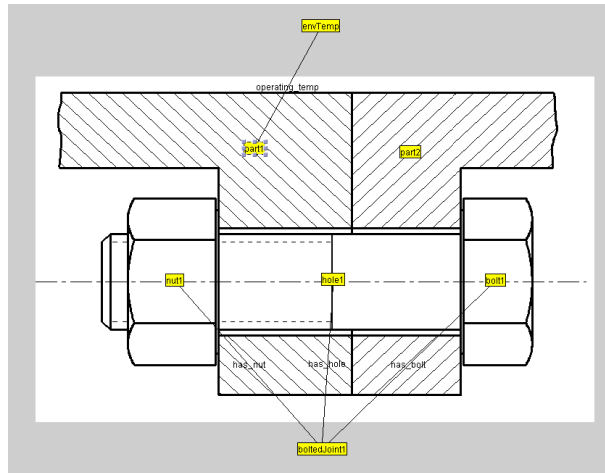


Figure 6: Closeup of the Designers' Workbench configuration panel

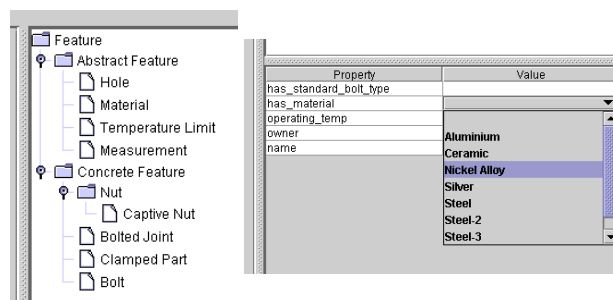


Figure 7: Closeups of the Designers' Workbench panels: the feature ontology (left), and properties of selected feature (right)

return results for the features that have sufficient values specified, so that only certain constraints will be checked. This allows designers to operate in an exploratory way, defining small parts of a configuration, checking them, and then gradually extending the configuration until it is complete.

6.3 Constraint rationales

Each constraint has an associated rationale (currently a short text string, but which in future may have more structure), and an (optional) URI for a source document explaining the rationale in more depth. When a constraint violation is reported, the designer is presented with a list of the features involved in the violation, the rationale, and link that can be clicked on to read the source document. In this way, the designer can learn more about the constraint, and decide if it is in fact appropriate. As the constraint checking proceeds, an experienced designer may decide to override the constraint.

7 Preliminary evaluation

In order to get feedback on the Designers' Workbench an informal interview was carried out with an engineer with six years experience at Rolls-Royce. The format of the interview was: an overview of the system was given by the developer, followed by the engineer attempting a series of tasks, and finally a discussion of the pros and cons of the system took place. The main findings were:

1. the interface was intuitive to use, although some details needed to be refined (for example, the exercise revealed some mismatches between the original ontology and the domain);
2. it was reasonably straightforward to add features to a design, although the engineer would have preferred textual items rather than icons. This could be implemented as an option that the user can set as required;
3. the engineer prefers that constraint checks are performed automatically, and an error message pops up, rather than the current arrangement of the user requesting that constraint checks be performed. Again, this could be an option that the user could enable or disable as required;
4. the engineer requested an additional feature, namely that links to previous designs or documents referring to similar features or issues be provided, enabling the user to find relevant past material quickly. Currently, previous designs or documents can be searched, but only by a text-based search;
5. the engineer did not currently use software that provides the same functionality as the Designers' Workbench.

Clearly, point 4 provides the most challenge. To find previous documents relating to particular features will require that the documents are annotated with reference to the same ontology as used by the Designers' Workbench. This means that authors of these documents must be provided with editors that will enable them to make these annotations as painlessly and as automatically as possible.

8 Future work

The Designers' Workbench is still at an early stage of development. In this section we describe some possible future avenues of research.

8.1 Propagation of values

A useful feature would be to reflect the designers' choices by maintaining domains for property values, and reducing them by removing values that are incompatible with the constraints. A problem arises here: a designer may have only tentatively selected a value for a particular property, and may deliberately wish to select an incompatible value for another property. A possible solution would be to display all the values in the domains for the designer to choose from, but highlight in some way the values that are incompatible with the currently selected values for other properties. The implementation of this feature will require the use of the constraint solving libraries of SICStus to maintain variable domains and perform propagation, rather than the simple predicate checking that is used currently.

8.2 Functional descriptions

The current version of Designers' Workbench does not deal explicitly with the functions of features. (Functions have been described using ontologies by Iwasaki et al. [IFVC93] and Kitamura et al. [KSNM02].) Nor does it deal with the various possible different states of features; for example, a shaft will have to operate successfully at different temperatures, pressures and rotational speeds. Adding functions to the Designers' Workbench will allow users to specify the intended functions of each feature, or group of features. Using an ontology of functions should allow for a fairly straightforward integration with the current system.

8.3 Design rationale storage (argumentation)

It would be very useful to capture and store the reasoning that the designer performs during the configuration process (design rationale capture). More details on processing design rationales can be found in [BS96] and [BW03]. Designers can examine current and past solutions to find out the reasons for particular decisions, why alternatives were discarded, and avoid repeating past mistakes.

8.4 Constraint input

To add a new constraint currently requires coding a query in RDQL, and a predicate in SICStus Prolog. This is quite a laborious task, and can only be done by a programmer. It would be useful if a new constraint could be formulated in an intuitive way, by selecting classes and properties from the ontology, and somehow combining them using a predefined set of operators. This would enable designers to have control over the definition and refinement of constraints, and presumably to be able to have greater trust in the results of constraint checks. We also intend to investigate alternative approaches to the underlying representation and querying of constraints. It is likely that the mixed RDQL/Prolog method may be replaced by a single framework, possibly Colan/CIF [GHP01].

8.5 Ontology change

Occasionally the available features described in the ontology may have to be increased (new parts or techniques become available), or decreased (obsolete parts removed). It is also possible that extensive changes may be made to naming conventions, or the arrangement of the feature class hierarchy. It is important to ensure that:

- previous designs can still be accessed using Designers' Workbench (possibly by recording the original ontology as part of each design);
- that existing constraints can be kept up to date.

A constraint that is defined on a class of features will automatically apply to a new subclass of that class. The problem of removals can be overcome by retaining the class, but flagging it in some way as obsolete. The most difficult situation will be the restructuring of a class hierarchy, which may require complex processing to ensure that existing constraints will still apply where necessary.

9 Summary

The Designers' Workbench allows designers to specify a configuration by selecting features from an ontology. Parameters of a feature can be specified by setting datatype property values, and the connections to other features are specified with objecttype properties. Constraints are defined on classes of features, but are applied to instances of features. A configuration can be checked for constraint violations at any time, even if some features are only partially specified. We have concentrated on assisting a human designer by checking constraints, rather than attempting to find solutions automatically. Constraint checking is performed by searching the configuration for applicable features, and passing the values of appropriate properties to a SICStus Prolog checker. A graphical display enables the designer to easily add new features, set property values, and perform constraint checks.

Acknowledgements

The authors would like to acknowledge the assistance of engineers and designers in the Transmissions and Structures division of Rolls-Royce plc, Derby, UK. The work was carried out as part of the EPSRC Sponsored Advanced Knowledge Technologies project, GR/NI5764, which is an Interdisciplinary Research Collaboration involving the University of Aberdeen, the University of Edinburgh, the Open University, the University of Sheffield and the University of Southampton.

References

- [Bar99] R. Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of Week of Doctoral Students (WDS99)*, pages 555–564, Prague, June 1999.
- [BHGS01] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*,

- number 2174 in Lecture Notes in Computer Science, pages 396–408, Vienna, September 2001. Springer-Verlag.
- [BOS90] J. Bowen, P. O’Grady, and L. Smith. A constraint programming language for life-cycle engineering. *Artificial Intelligence in Engineering*, 5(4):206–220, 1990.
- [Bro98] D. C. Brown. Defining configuring. *AI EDAM*, 12:301–305, September 1998.
- [Bro02] D. C. Brown. Functional, behavioral and structural features. In J. C. Borg and P. Farrugia, editors, *Proc. KIC5: 5th IFIP WG5.2 Workshop on Knowledge Intensive CAD*, Malta, July 2002.
- [BS96] S. Buckingham Shum. Design argumentation as design rationale. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, pages 95–128. Marcel Dekker, Inc., 1996.
- [BW03] R. Bracewell and K. Wallace. A tool for capturing design rationale. In *ICED03, 14th International Conference on Engineering Design*, pages 185–186, Stockholm, Sweden, 2003.
- [CJB99] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins. What are ontologies and why do we need them? *IEEE Intelligent Systems*, 14(1):20–26, Jan/Feb 1999.
- [DAM] The darpa agent markup language homepage. <http://www.daml.org/>.
- [GHP01] P. Gray, K. Hui, and A. Preece. An expressive constraint language for semantic web applications. In *IJCAI-01 Workshop on E-Business and the Intelligent Web*, pages 46–53, Seattle, USA, August 2001.
- [IFVC93] Y. Iwasaki, R. Fikes, M. Vescovi, and B. Chandrasekaran. How things are intended to work: Capturing functional knowledge in device design. In *IJCAI-93*, pages 1516–1522, 1993.
- [JEN] Jena - a semantic web framework for java. <http://jena.sourceforge.net/index.html>.
- [JGR] The home page of jgraph. <http://www.jgraph.com/>.
- [JM03] U. Junker and D. Mailharro. The logic of ilog (j)configurator: Combining constraint programming with a description logic. In *Proceedings of IJCAI’03 Workshop on Configuration*, 2003.
- [KSNM02] Y. Kitamura, T. Sano, K. Namba, and R. Mizoguchi. A functional concept ontology and its application to automatic identification of functional structures. *Advanced Engineering Informatics*, 16(2):145–163, April 2002.
- [Kum92] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, pages 32–40, Spring 1992.
- [Lab03] F. Laburthe. Constraints over ontologies. In *Proceedings of CP2003*, 2003.

- [LC02] L. Lin and L.-C. Chen. Constraints modelling in product design. *Journal of Engineering Design*, 13(3):205–214, September 2002.
- [LFB96] Jinxin Lin, M. Fox, and T. Bilgic. A requirement ontology for engineering design. *Concurrent Engineering: Research and Applications*, 4(4):279–291, September 1996.
- [McG02] D. L. McGuinness. Ontologies come of age. In D. Fensel, J. Hendler, H. Lieberman, and W. Wahlster, editors, *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2002.
- [MF89] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 1395–1401, San Francisco, CA, 1989. Morgan Kaufmann.
- [MF90] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI-90: The Eighth National Conference on Artificial Intelligence*, pages 25–32. MIT Press, 1990.
- [MW98] D. L. McGuinness and J. R. Wright. Conceptual modelling for configuration: A description logic-based approach. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12:333–344, 1998.
- [SBF98] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: Principles and methods. *Data Knowledge Engineering*, 25(1–2):161–197, 1998.
- [SF96] D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. In George F. Luger, editor, *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996, 1996.
- [SFH98] M. Stumptner, G. E. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12:307–320, 1998.
- [Smi95] B. M. Smith. A tutorial on constraint programming. Technical Report 95.14, School of Computer Studies, University of Leeds, April 1995.
- [Swe03] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 2003.