Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2008), *22*, 325–343. Printed in the USA.

doi:10.1017/S089006040800022X

Constraint capture and maintenance in engineering design

SURAJ AJIT,[1] DEREK SLEEMAN,[1] DAVID W. FOWLER,[1] AND DAVID KNOTT[2]

[1]Department of Computing Science, University of Aberdeen, Aberdeen, Scotland
[2]Rolls-Royce plc, Derby, United Kingdom

(RECEIVED April 30, 2007; ACCEPTED May 9, 2008)

Abstract

The Designers' Workbench is a system developed by the Advanced Knowledge Technologies Consortium to support designers in large organizations, such as Rolls-Royce, to ensure that the design is consistent with the specification for the particular design as well as with the company's design rule book(s). In the principal application discussed here, the evolving design is described using a jet engine ontology. Design rules are expressed as constraints over the domain ontology. Currently, to capture the constraint information, a domain expert (design engineer) has to work with a knowledge engineer to identify the constraints, and it is then the task of the knowledge engineer to encode these into the Workbench's knowledge base. This is an error-prone and time-consuming task. It is highly desirable to relieve the knowledge engineer of this task, so we have developed a system, ConEditor+, that enables domain experts themselves to capture and maintain these constraints. Further, we hypothesize that to appropriately apply, maintain, and reuse constraints, it is necessary to understand the underlying assumptions and context in which each constraint is applicable. We refer to them as "application conditions," and these form a part of the rationale associated with the constraint. We propose a methodology to capture the application conditions associated with a constraint and demonstrate that an explicit representation (machine interpretable format) of application conditions (rationales) together with the corresponding constraints and the domain ontology can be used by a machine to support maintenance of constraints. Support for the maintenance of constraints includes detecting inconsistencies, subsumption, redundancy, fusion between constraints, and suggesting appropriate refinements. The proposed methodology provides immediate benefits to the designers, and hence, should encourage them to input the application conditions (rationales).

Keywords: Application Conditions; Capture; Constraints; Design; Maintenance; Rationales

1. INTRODUCTION

"Knowledge management has been identified as one of the key enabling technologies for distributed engineering enterprises in the 21st century. Central to the application and exploitation of knowledge in engineering is the engineering design process" (McMahon et al., 2004). The Advanced Knowledge Technologies[1] Project has identified six major challenges involving the acquisition, modeling, reuse, retrieval, publishing, and maintenance of knowledge. The challenges relevant in the context of the work reported in this article are knowledge acquisition and maintenance, where the knowledge here refers to the design rules and rationales in engineering design, represented against the domain ontology. Knowledge acquisition is about extracting knowledge from sources of expertise and transferring it to a knowledge base (KB). Knowledge acquisition is well known to be a "critical bottleneck" in knowledge-based system (KBS) development. The traditional approach to knowledge acquisition is mainly an interactive process involving the domain expert and knowledge engineer. This approach can be tedious, time consuming, and error prone, especially if the knowledge engineer is unfamiliar with the domain. Knowledge maintenance is concerned with making necessary changes to existing knowledge bases so that redundant and inappropriate information is removed. This normally involves the following activities:

1. Verification and validation of knowledge based systems: Verification and validation of the content of knowledge repositories is at the heart of knowledge maintenance. Verification is a process of ensuring that the knowledge base is consistent and complete within itself. Validation is the process of determining if a KBS meets its users' requirements (Meseguer & Preece, 1995).
2. Updating/refining of knowledge bases: The challenge is to keep the knowledge repository functional. This may

Reprint requests to: Suraj Ajit, Department of Computing Science, University of Aberdeen, Aberdeen, Scotland. E-mail: surajajit@yahoo.com

[1] Advanced Knowledge Technologies (AKT), accessed online at http://www.aktors.org on August 29, 2006.

involve the regular updating/refining of content as it changes (e.g., as price lists are revised). But it may also involve a deeper analysis of the knowledge content. Some content has a considerable longevity, whereas other knowledge dates very quickly. If a repository of knowledge is to remain active over a period of time, it is essential to know which parts of the knowledge base must be discarded and under what conditions.
3. Dealing with the obsolescence of knowledge: Certain sections of the knowledge may be based on assumptions/conditions that later become untrue. One has to identify and shelve/remove such sections, when necessary. When the knowledge base is updated, a lot of redundant knowledge can be accumulated in the knowledge base.

The issues faced in KB maintenance within engineering were first raised by the XCON configuration system at Digital Equipment Corporation (Soloway et al., 1987; Barker & O'Connor, 1989). "Initially it was assumed that knowledge-based systems could be maintained by simply adding new elements or replacing existing elements. However this simplicity proved to be illusory as indicated by the experience of R1/XCON" (Coenen, 1992).

Engineering design is constraint oriented, and much of the design process involves the recognition, formulation, and satisfaction of constraints (Serrano & Gossard, 1992; Lin & Chen, 2002). The engineering design process has an evolutionary and iterative nature as designed artifacts often develop through a series of changes before a final solution is achieved. A common problem encountered during the design process is that of knowledge (e.g., constraint) evolution, which may involve the identification of new constraints or the modification or deletion of existing constraints. The reasons for such changes include development in the technology, changes to improve performance, changes to reduce development time and costs. Typically, maintenance involves various issues and problems:

1. Original experts are unlikely to be available: The transient nature of modern organizations and workforces, the rapid flow of knowledge, and experience out of companies because of staff leaving make it difficult for new designers to properly use stored design knowledge and subsequently to maintain it.
2. Insufficient documentation provided: Several constraints may be applicable only in particular contexts. These contexts are often implicit to the designer formulating them but are not documented. Also, many constraints are based on assumptions that have become untrue subsequently. These assumptions are often not made explicit.
3. Maintenance is time consuming and complex: Maintenance of constraints in an engineering design environment is a complicated process that can be complicated and time consuming to do manually. Thus, there is a pressing need for tools to support maintenance of this kind of knowledge.
4. The evolutionary nature of constraints establishes the need to constantly update, revise, and maintain them. One needs to identify all the constraints that require modification. Also, one needs to make sure that the knowledge base is consistent after making any changes.

In addition, verification in KBSs plays a very important role. As we automate more and more processes, the need for verification becomes even more critical. Many automated processes perform incorrectly for a long time, as no person is responsible for checking the process (Hicks, 2003). As the KB evolves, constant addition and revision of rules can result in many redundancies. It is important to prevent or at least reduce the number of redundant rules in a KB. Removing or reducing the redundancy in a KB will make it easier to maintain the KB. Moreover, design often involves the reuse and modification of past designs. For example, research has identified that up to 90% of all design activities are based on the variants of existing designs (Fletcher & Gu, 2005). Knowing the contexts in which certain design rules are applicable becomes extremely important for design maintenance and reuse.

1.1. Constraints, assumptions, and contexts as design rationales

Constraints are continually being added, deleted, and modified throughout the development of a new device. Design begins with a functional specification of the desired product: a description of properties and conditions that the product should satisfy (i.e., constraints). Constraints themselves form a rationale associated with the design decisions taken by designers. A typical rationale is of the form: "A component X exists in the design because of the need to satisfy constraint Y." The ability to capture and use this type of design rationale in concurrent engineering has been referred to as design rationale management by Bahler and Bowen (1992), who describe a constraint-based design advice system that generates machine-generated suggestions to support coordination among multiple design engineers. The Designers' Workbench (Fowler et al., 2004) provides similar functionality by checking if the design satisfies all the relevant constraints, providing details of the violated constraints and enabling the designers to resolve them.

Constraints themselves may be formulated based on a number of assumptions, and may be relevant only in certain contexts. Designers often tend to assume "normal" situations (Brown, 2006). They tend to make assumptions about the match between the current design situation and one where their chosen technique worked well before by assuming that some key detail is relevant or irrelevant. These assumptions are not deliberate, but form the tacit knowledge underlying expert skill. To support maintenance of designs it is important to make these assumptions visible. We need to find ways to capture the assumptions and contexts as part of the rationale

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

associated with a constraint. We refer to this type of rationale as the *application conditions* associated with a constraint. A recent article (Hooey & Foyle, 2007) reported on the requirements for design rationale capture tool to support all the design phases of NASA's complex systems. They stressed the need to capture the assumptions and constraints as the rationale for a given design element, particularly in the conceptual design phase. This article describes how this information is rarely captured in a systematic and usable format because there are no tools that adequately facilitate and support the capture and use of this critical information. An example quoted in the article is: "The minimum volume for the Crew Exploration Vehicle cockpit is based on an assumption of a specific crew size." The above example is clearly a constraint (minimum volume for the Crew Exploration Vehicle), together with its application condition (specific crew size). Also, if a design element or a constraint is modified, there is no easy way to propagate that change to understand the implications and consequences of those changes. Thus, it is important to capture information pertaining to when a particular section of the design knowledge is applicable and also enable machines to *use* this information to support maintenance. The following section describes the research aims and hypothesis of the work reported in this article.

1.2. Research aims and hypothesis

Enabling domain experts to maintain knowledge in a KB system has long been an ideal for the knowledge engineering community (Bultman et al., 2000). This article identifies a situation where it is highly desirable to eliminate the knowledge engineer from doing a laborious, error-prone and time-consuming task. The article reports on a system ConEditor+ that we have developed to enable domain experts themselves to capture and maintain constraints. Further, we hypothesize that it is important to capture the assumptions and context in which a constraint is applicable in a machine-interpretable format, and that this rationale information (referred to as application conditions) together with the constraints and the domain ontology can be used by a machine to support the maintenance of constraints. By supporting the maintenance of the constraints we mean that an explicit representation of application conditions together with the constraints and the domain ontology could help a machine in reducing the number of inconsistencies and also in performing appropriate refinements of subsumption, redundancy, and fusion between pairs of constraints. Design rationale systems usually capture the information in a human readable format. Although the information may have some structure, the information cannot be understood, interpreted, and used by machines to provide immediate benefits to the designers. Design rationales are also often difficult to retrieve and, hence, rarely used. We aim to capture application conditions as rationales together with the constraints and enable the system to use this information together with the domain ontology to detect inconsistencies and suggest appropriate refinements between constraints. The main research question we plan to address is as follows: "Could an explicit representation of application conditions together with the constraints and the domain ontology help a machine in: a) reducing the number of inconsistencies and b) detecting subsumption, redundancy, fusion and suggesting appropriate refinements between pairs of constraints? In other words, could an explicit representation of application conditions together with the constraint and the domain ontology be used by a machine to support the maintenance of constraints?" The following section describes the layout of this article.

1.3. Layout

The context for the principal system reported here, ConEditor+ (Ajit et al., 2005), is the Designers' Workbench that has been developed to enable a group of designers to produce cooperatively a component that conforms to the component's overall specifications and the company's design rule book(s). Section 2.1 provides an introduction to the Designers' Workbench and the motivation for the development of ConEditor+. Section 2.2 gives an overview of the system ConEditor+. Section 2.3 summarizes our proposed approach. Section 3 describes the conceptual design by considering the kite design domain. Section 4 describes the implementation of our proposed approach. We discuss the evaluation and results in Section 5, followed by a review of relevant work in Section 6. The conclusions and plans for future work follow in Section 7.

2. CONSTRAINT CAPTURE AND MAINTENANCE IN ENGINEERING DESIGN: A PROPOSAL

2.1. Introduction to the Designers' Workbench

Designers in Rolls-Royce, as in many large organizations, work in teams. Thus, it is important when a group of designers are working on aspects of a common project, that the subcomponent designed by one engineer is consistent with the overall specification, and with those designed by other members of the team. In addition, all designs have to be consistent with the company's design rule book(s). Making sure that these various constraints are complied with is a complicated process, so we have developed the Designers' Workbench which seeks to support these activities.

The Designers' Workbench (Fig. 1) uses an ontology (Gruber, 1995) to describe the element to be designed. Design rules are expressed as constraints over the domain ontology. The system supports human designers by checking that their configurations satisfy both physical and organizational constraints. Configurations are composed of features, which can be geometric or nongeometric, physical or abstract. A graphical user interface (GUI) enables the designer to easily add new features, assign property values, and perform constraint checks. If a constraint is violated, the affected features are

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

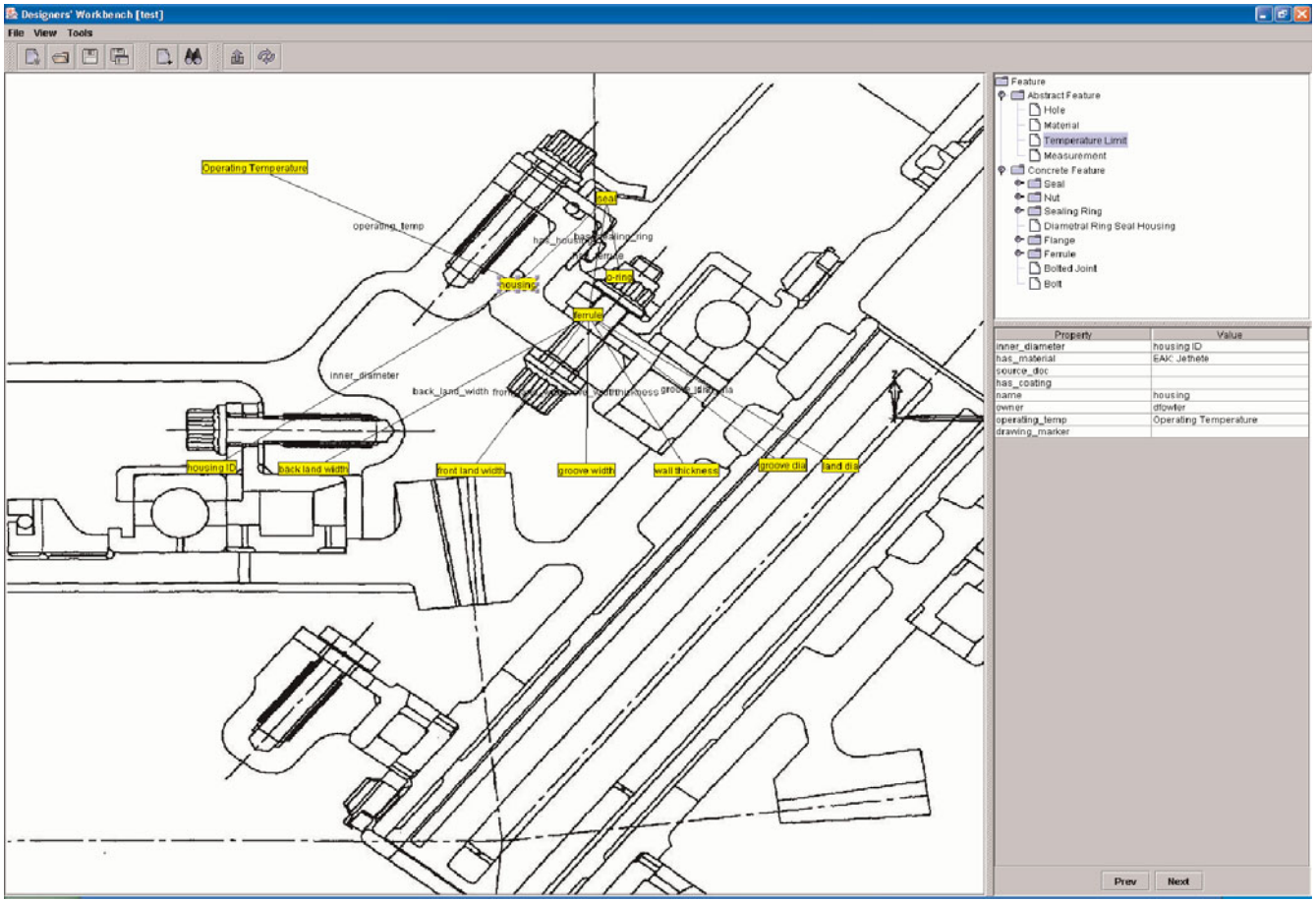

Fig. 1. Screenshot of the Designers' Workbench. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

highlighted and a report is generated. The report gives the designer a short description of the constraint that is violated, the features affected by that violation, and a link to the source document. The designer can resolve the violations by adjusting the property values of the affected features. On selecting the affected feature from the ontology, a table is displayed with the corresponding properties and values. These property values can then be adjusted, and this usually resolves the constraint violations. More details about this system can be found in (Fowler et al., 2004).

2.1.1. Capturing the knowledge in the design rule books

As noted above, the Designers' Workbench needs access to the various constraints, including those inherent in the company's design rule books. Currently, to capture this information, a design engineer (domain expert) works with a knowledge engineer to identify the constraints, and it is then the task of the knowledge engineer to encode these into the Workbench's KB. This is an error-prone and time-consuming task. As the constraints are explained succinctly in the design rule books, a nonexpert in the field can find it very difficult to understand the context and formulate constraints directly from the design rule books, so a design engineer has to help the knowledge engineer in this process. Most design rules are specified using technical drawings. Adding a new constraint into the Designers' Workbench's KB requires coding a query in RDQL language (Seaborne, 2004) and a predicate in SICStus[2] Prolog.

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 help engineers to formulate constraints themselves and relieve the programmer of that task. This would also enable designers to have greater control over the definition and refinement of constraints, and presumably, to have greater trust in the results of the constraint checking process. This led to the development of a system, called ConEditor+, which enables a domain expert to capture and maintain constraints. ConEditor+ is explained further in the next section.

2.2. ConEditor+

ConEditor+ has been designed to enable domain experts to capture and maintain constraints. ConEditor+'s GUI is shown in Figure 2. A constraint expression can be created by selecting entities from a domain ontology and combining them with a predefined set of keywords and operators from the high level constraint language, CoLan (Bassiliades & Gray, 1995; Gray et al., 2001). CoLan has features of both first-order logic and functional programming. CoLan is designed to enable scientists and engineers to express constraints.

An example of a simple constraint expressed in CoLan, against a domain ontology (a jet engine ontology) used by the Designers' Workbench is as follows[3]:

constrain each f **in** ConcreteFeature **to have**
max_operating_temp(has_material(f)) >= operating_temp(f)

This constraint states that for every instance of the class ConcreteFeature, the value of the maximum operating temperature of its material must be greater than or equal to the environmental operating temperature. We now look at how the example constraint can be formulated using ConEditor+.

ConEditor+'s GUI (Fig. 2) essentially consists of six components:

1. Keywords panel: The keywords panel consists of a list of keywords from the CoLan language. In the example considered, the keywords *constrain each*, *in*, *to have* are selected from this panel. A single mouse click on a keyword appends it to the text area in the result panel. Alternatively, clicking the "Add" button, after selecting the keyword from the panel, appends the keyword to the text area in the result panel.
2. Menu bar: The menu bar contains a list of menus and submenus with operations for loading, editing, deleting, searching, saving constraints, and performing syntax checks.
3. Functions panel: The functions panel consists of six buttons (Erase, Create Table, Submit, Query, Open, Save) that can be clicked to perform some of the frequently used operations from the menu bar.
4. Taxonomy panel: The taxonomy panel lists all the top level classes (i.e., classes having its parent as Thing in OWL ontology) in the domain ontology together with their subclasses, properties (both object and datatype), and properties of the range classes as a taxonomy. Each class or object property can be expanded by a double mouse click to list all the subclasses and properties below it in the taxonomy. Nodes represented by letter "P" denote properties, whereas the remaining nodes denote classes. Selecting a node using the mouse and clicking the Add button appends the entity represented by the node to the constraint expression being formed in the result panel. In the example considered, the entities ConcreteFeature, max_operating_temp, has_material, and operating_temp are selected from this panel.
5. Tool bar: The tool bar displays the operators (arithmetic, relational, and logical) and delimiters. In the example considered, the operator $\geq$ and the delimiters (',') are selected from the tool bar. Again, a single

[2] Swedish Institute of Computer Science, version 3.10, accessed online at http://www.sics.se/sicstus/ on August 29, 2006.

[3] The naming convention of the properties defined in the domain ontology could be changed appropriately to make the constraint more readable. As an example, the constraint above could be expressed alternatively as **constrain each** f **in** ConcreteFeature **to have** max_operating_temp_of(material_of(f)) $\geq$ operating_temp_of(f).

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

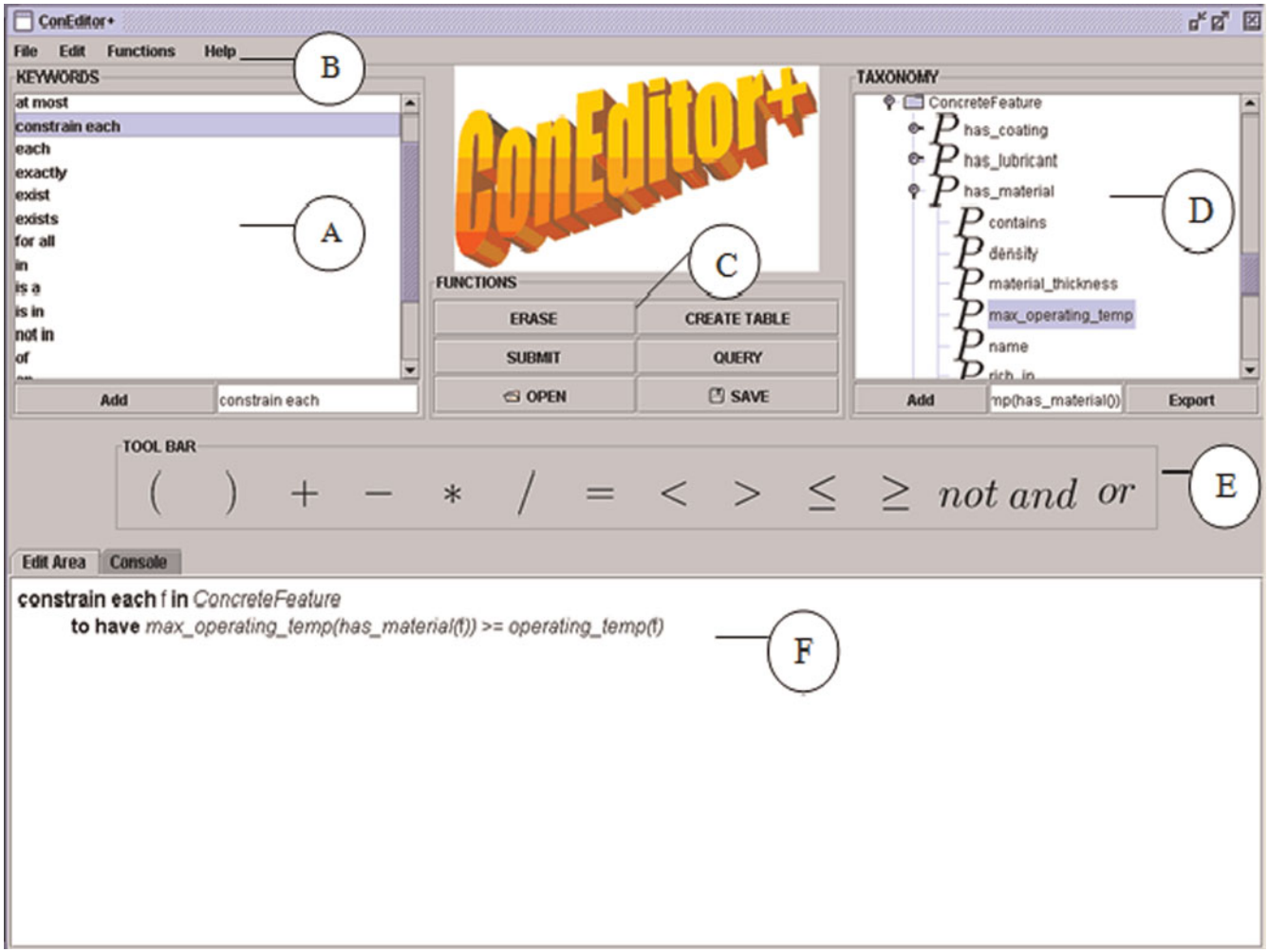

Fig. 2. A screenshot of ConEditor+. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

mouse click on the selected operator appends the operator to the text area in the result panel.

6. Result panel: The result panel consists of a text area, displaying the constraint expression formulated by the user and any output messages (e.g., syntax error message) from ConEditor+. This panel consists of two tabs, namely, the Edit Area and the Console, that display the constraint expression formulated by the user and the output messages from the system, respectively.

ConEditor+ provides a mechanism to input constraints in the form of tables also. When a constraint is modified and saved, ConEditor+ normally stores the modified constraint as a new version along with the original constraint. The rationale for storing different versions of a constraint is to enable designers to study the constraint evolution (Goonetillake & Wikramanayake, 2004). Each constraint is allocated a unique identification number (ID) that includes its version number. The system provides facilities to retrieve constraints using keyword-based searches, for example, search and retrieve all the constraints containing the specified keywords or the constraint associated with a specified ID.

2.3. Proposed approach

Because of the restricted availability of Rolls-Royce designers and because it is a simpler domain, we used the kite domain for our initial study (Yolen, 1976; Streeter, 1980; Eden, 1998; AKA, 2006; CEKS, 2006; Leigh, 2006; Lords, 2006; Wardley, 2006). For a successful kite design, one has to make sure that the design complies with all the appropriate rules/constraints.

Figure 3 shows the diagram of a flat diamond kite with all its basic parts. Consider the following constraint and its application condition:

Constraint: "The density of the cover material of the kite must be greater than 0.5 ounce per square inch."

Application condition: "This is applicable only when there is a requirement to produce low cost kites for beginners. Kites for experts have lighter materials that are of higher quality and hence costlier."

This example shows the application condition specifies the context in which the constraint is applicable. We believe that it is important to make the application conditions *explicit* so that it can be used to apply constraints appropriately and also be used in the reuse and maintenance of constraints. Often, the information of application conditions is implicit to the person who formulates the constraint. The assumptions/conditions on which a constraint is based may no longer be true, and in such cases it becomes necessary to deactivate or remove those constraints from the KB.

Although design rationales can provide a lot of information about the reasoning involved in making a design decision, rationales are extremely hard to collect mainly because the process is very intrusive and requires a lot of the designers' time. If rationales are useful to the designers, there is a greater incentive for designers to assist in the capture of the information, particularly if the designer who is recording it can immediately use the rationale. As Grudin (1996) and Brown (2006) have pointed out, there cannot be a disparity between who invests effort in a groupware system and who benefits. No designer can be expected to altruistically enter quality design rationale solely for the possible benefit of a possibly unknown person at an unknown point in the future for an unknown task. There must be immediate value. In addition, knowing how the information will be used provides guidance about what information should be captured and how it should be represented. Thus, it is important to concentrate on the immediate use of such information (Burge & Brown, 2003). Representation of the rationales in a machine-interpretable form would enable systems to use them together with the constraints and the domain ontology to detect inconsistencies, redundancy, subsumption, and fusion and to suggest appropriate refinements between constraints.

Our proposed approach is to capture the application conditions together with the constraint and use that information together with the domain ontology to support the maintenance of constraints. To tackle the various maintenance issues/problems effectively, our proposed solution is summarized as follows:

1. Capture the assumptions and context of each constraint, in a machine-interpretable form, as an application condition (rationale).
2. Use the application condition together with the constraint and the corresponding domain ontology to detect inconsistencies, redundancies, subsumptions, and fusions between constraints and suggest appropriate refinements (described in greater detail in Section 3).

The next two sections describe the conceptual design and implementation of our proposed approach with examples.

3. CONCEPTUAL DESIGN

ConEditor+ captures both the constraints and the application conditions in the same language, CoLan. Representation of a sample constraint with its application condition in CoLan is shown below:

constrain each k **in** Kite **such that** *has_type*(*k*)
= "*Flat*" *and has_shape*(k) = "*Diamond*" **to have**
tail_length(has_tail(k)) = 7 ∗ spine_length(has_spine(k))

In this constraint, the application condition (in italics) is introduced by the clause "such that." This constraint states that "For every instance of the class Kite, when the type of the

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

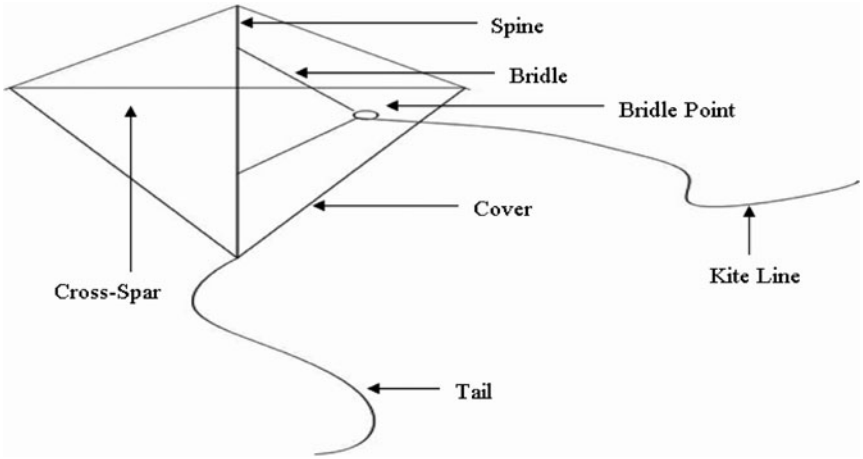

Fig. 3. The basic parts of a flat diamond kite.

kite is flat and shape of the kite is diamond, the length of the tail of the kite needs to be seven times the length of the spine of the kite." To make it clearer, we divide a constraint represented in CoLan into three parts: antecedent, application condition, and consequent. Thus, a constraint is represented by the following general structure:

constrain each x_1 **in** C_1
each x_2 **in** C_2 (Antecedent)
. . . . . .

such that $P_1(x_1)$
$P_2(x_2)$ (Application Condition)
. . . . . .

to have $R_1(x_1)$
$R_2(x_2)$ (Consequent)
. . . . . .

The representation of the CoLan constraint described above, in first-order logic, is as follows:

$$\forall k[(\text{Kite}(k) \wedge (\text{has\_type}(k) = \text{"Flat"}) \wedge (\text{has\_shape}(k) = \text{"Diamond"})) \rightarrow (\text{tail\_length}(\text{has\_tail}(k)) = 7 * \text{spine\_length}(\text{has\_spine}(k)))]$$

A constraint in CoLan, in general, can be represented by a first order-logic sentence as

$$S \equiv \forall x_1, \ldots, x_n[(C_1(x_1) \wedge \ldots . \wedge C_n(x_n) \wedge P_1(x_1, \ldots, x_n) \wedge \ldots . \wedge P_m(x_1, \ldots, x_n)) \rightarrow R(x_1, \ldots, x_n)]$$

where S is a sentence; $x_1, \ldots, x_n$ are variables; $C_1, \ldots, C_n$ are classes; and $P(x_1, \ldots, x_n)$, $P_m(x_1, \ldots, x_n)$, $R(x_1, \ldots, x_n)$ represent predicates/properties.

There are a number of ways in which we can use the information inherent in application conditions together with the constraint and the associated ontology to enable the maintenance of constraints. We propose four main types of knowledge refinement rules, namely, redundancy, subsumption, contradiction, and fusion between pairs of constraints. These rules are applied between all possible pairs of constraints. The knowledge refinement rules are described below with examples from the kite domain. A formal notation in first-order logic for each knowledge refinement rule together with the logical proof can be found in Ajit (2008).

3.1. Redundancy

3.1.1. Duplication

(i) **constrain each** c **in** ConventionalSledKite **such** that has_level(c) = "beginner" **to have** density (has_material) (has_cover(c))) < 0.5
(ii) **constrain each** c **in** ConventionalSledKite **such that** has_level(c) = "beginner" **to have** density(has_material) (has_cover (c))) < 0.5

By comparing these two constraints, one can infer that constraints (i) and (ii) are identical.

3.1.2. Class equivalence

(iii) **constrain each** c **in** ConventionalSledKite **such that** has_level (c) = "beginner" **to have** density(has_material(has_cover(c))) < 0.5
(iv) **constrain each** t **in** TraditionalSledKite **such that** has_level(t) = "beginner" **to have** density(has_material(has_cover(t))) < 0.5

As ConventionalSledKite is an equivalent class to TraditionalSledKite in the domain ontology, one can infer that the constraint (iii) is equivalent to constraint (iv).

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

3.1.3. Property equivalence

(v) **constrain each** c **in** ConventionalSledKite **such that** has_level(c) = "beginner" **to have** density(has_material(has_cover(c))) < 0.5
(vi) **constrain each** c **in** ConventionalSledKite **such that** has_class(c) = "beginner" **to have** density(has_material(has_cover(c))) < 0.5

As has_level is an equivalent property to has_class in the domain ontology, one can infer that the constraint (v) is equivalent to constraint (vi).

ConEditor+ reports all such redundancies to the domain expert and suggests that they be removed.

3.2. Subsumption

3.2.1. Subsumption via subclass

(vii) **constrain each** s **in** SledKite **such that** has_size(s) = "standard" **to have** kite_line_strength(has_kite_line(s)) >= 15
(viii) **constrain each** c **in** ConventionalSledKite **such that** has_size(c) = "standard" **to have** kite_line_strength(has_kite_line(c)) >= 15

As ConventionalSledKite is a subclass of SledKite in the domain ontology, one can infer that the constraint (vii) subsumes constraint (viii). The domain expert is notified of this fact and ConEditor+ suggests that the domain expert considers removing or deactivating constraint (viii).

3.2.2. Subsumption via application condition

(ix) **constrain each** s **in** SledKite **such that** has_size(s) = "standard" **or** has_size(s) = "large" **to have** kite_line_strength(has_kite_line(s)) >= 15
(x) **constrain each** s **in** SledKite **such that** has_size(s) = "standard" **to have** kite_line_strength(has_kite_line(s)) >= 15

By comparing the two constraints above, one can infer that the constraint (ix) subsumes constraint (x). The domain expert is notified of this fact and ConEditor+ suggests that the domain expert considers removing or deactivating constraint (x).

3.2.3. Subsumption via conjunction

(xi) **constrain each** s **in** SledKite **such that** has_size(s) = "standard" **to have** kite_line_strength(has_kite_line(s)) >= 15 **and** has_cord_length(s) > 21
(xii) **constrain each** s **in** SledKite **such that** has_size(s) = "standard" **to have** kite_line_strength (has_kite_line(s)) >= 15

Again, one can infer that the constraint (xi) subsumes constraint (xii). The domain expert is notified of this fact and ConEditor+ suggests that the domain expert considers removing or deactivating constraint (xii).

3.3. Contradiction

(xiii) **constrain each** k **in** Kite **such that** has_type(k) = "stunt" **to have** kite_line_strength(has_kite_line(k)) > 30
(xiv) **constrain each** k **in** Kite **such that** has_type(k) = "stunt" **to have** kite_line_strength(has_kite_line(k)) < 25

By comparing the two constraints above, one can infer that the constraint (xiii) contradicts constraint (xiv). The domain expert is notified of this fact and ConEditor+ suggests that the domain expert takes an appropriate action (modify/delete).

3.4. Fusion

3.4.1. Fusion via class

(xv) **constrain each** c **in** ConventionalSledKite **such that** has_wind_condition(c) = "moderate" **to have** has_bridle_attachment_angle(c) < 40
(xvi) **constrain each** m **in** ModernSledKite such that has_wind_condition(m) = "moderate" **to have** has_bridle_attachment_angle(m) < 40

If ConventionalSledKite and ModernSledKite are the only two subclasses of SledKite in the domain ontology and if every instance of SledKite is an instance of either ConventionalSledKite or ModernSledKite then the constraints (xv) and (xvi) can be fused together and replaced by the constraint (xvii) as follows:

(xvii) **constrain each** s **in** SledKite **such that** has_wind_condition(s) = "moderate" **to have** has_bridle_attachment_angle(s) < 40

3.4.2. Fusion via application condition

(xviii) **constrain each** j **in** JapaneseKite **such that** has_wind_condition(j) = "strong" **to have** has_bridle_point_distance(j) > 3 * surface_area(has_cover(j))
(xix) **constrain each** j **in** JapaneseKite **such that** has_type(j)="stunt" **to have** has_bridle_point_distance(j) > 3 * surface_area(has_cover(j))

The constraints above can be fused together by using "or" between the application conditions, that is, the constraints (xviii) and (xix) can be fused together and replaced by the constraint (xx) as follows:

(xx) **constrain each** j **in** JapaneseKite **such that** has_wind_condition(j) = "strong" **or** has_type(j) = "stunt" **to have** has_bridle_point_distance(j) > 3 * surface_area(has_cover(j))

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

3.4.3. Fusion via conjunction

(xxi) **constrain each** j **in** JapaneseKite **such that** has_wind_condition(j) = "strong" **to have** has_bridle_point_distance(j) > 3 ∗ surface_area(has_cover(j))

(xxii) **constrain each** j **in** JapaneseKite **such that** has_wind_condition(j) = "strong" **to have** kite_line_strength(has_kite_line(j)) >= 15

The constraints above can be fused together by using "and," that is, the constraints (xxi) and (xxii) can be fused together and replaced by the constraint (xxiii) as follows:

(xxiii) **constrain each** j **in** JapaneseKite **such that** has_wind_condition(j) = "strong" **to have** has_bridle_point_distance(j) > 3 ∗ surface_area(has_cover(j)) **and** kite_line_strength(has_kite_line(j)) >= 15

In all cases, ConEditor+ makes suggestions but allows the domain expert to decide on what action, if any, to take. In all the examples above, we have considered universally quantified constraints involving a single variable; these types of expressions are common in our KB. However, more complex first-order logic expressions involving existential quantifiers and many variables or a combination of both existential and universal quantifiers can also be expressed in CoLan using ConEditor+.

Thus, we have described four main types of knowledge refinement rules among constraint pairs with all the refinements (except contradiction) having subtypes:

1. Redundancy: (a) duplication, (b) class equivalence, (c) property equivalence
2. Subsumption: (a) via subclass, (b) via application condition, (c) via conjunction
3. Contradiction
4. Fusion: (a) via class, (b) via application condition, (c) conjunction

Knowledge refinement rules can be combined and applied together to a pair of constraints. For an example, consider the following constraints:

(E1) **constrain each** s **in** SledKite **such that** has_type(s) = "stunt" **or** has_wind_condition(s) = "strong" **to have** kite_line_strength(has_kite_line(s)) > 30

(E2) **constrain each** c **in** ConventionalSledKite **such that** has_type(c) = "stunt" **to have** kite_line_strength (has_kite_line(c)) < 25

By comparing the constraints (E1) and (E2), we have the following:

a. ConventionalSledKite is a subclass of SledKite in the domain ontology.
b. The application condition of constraint (E2) is subsumed by the application condition of constraint (E1).
c. The consequent of constraint (E1) contradicts the consequent of constraint (E2).

Hence, one can infer that the constraint (E1) contradicts constraint (E2) and makes the KB inconsistent. The domain expert is notified of this fact and ConEditor+ suggests that the domain expert takes an appropriate action (modify/delete). In the example above, we have applied a combination of the following knowledge refinement rules: subsumption via subclass, subsumption via application condition, contradiction. ConEditor+ applies such combinations of knowledge refinement rules to detect inconsistencies and suggest appropriate refinements among constraint pairs. ConEditor+' s algorithm to determine the order in which refinement rules are applied is outlined next.

Consider a pair of constraints A and B. Let the antecedents be represented by AN_a and AN_b, application conditions by AC_a and AC_b, and consequents by C_a and C_b for constraints A and B, respectively.

Step 1: Check for redundancy (whether A is identical to B):
If AN_a not equal/equivalent to AN_b then go to step 2a.
If AC_a not equal/equivalent to AC_b then go to step 2a.
If C_a equal/equivalent to C_b then conclude redundancy, notify user (domain expert), suggest refinement action(s) and exit.

Step 2a: Check for subsumption (whether A subsumes B):
If AN_a not equal/equivalent/subsumes AN_b then go to step 2b.
If AC_a not equal/equivalent/subsumes AC_b then go to step 2b.
If C_a equal/equivalent/subsumes C_b then conclude subsumption, notify user (domain expert), suggest refinement action(s) and exit.

Step 2b: Check for subsumption (whether B subsumes A):
If AN_b not equal/equivalent/subsumes AN_a then go to step 3a.
If AC_b not equal/equivalent/subsumes AC_a then go to step 3a.
If C_b equal/equivalent/subsumes C_a then conclude subsumption, notify user (domain expert), suggest refinement action(s) and exit.

Step 3a: Check for contradiction (whether A contradicts B):
If AN_a not equal/equivalent/subsumes AN_b then go to step 3b.
If AC_a not equal/equivalent/subsumes AC_b then go to step 3b.
If C_a contradicts C_b then conclude contradiction, notify user (domain expert), suggest refinement action(s) and exit.

Step 3b: Check for contradiction (continued):
If AN_b not equal/equivalent/subsumes AN_a then go to step 4a.

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

If AC_b not equal/equivalent/subsumes AC_a then go to step 4a.
If C_a contradicts C_b then conclude contradiction, notify user (domain expert), suggest refinement action(s) and exit.

Step 4a: Check for fusion (whether A and B can be fused):
If AN_a not equal/equivalent to AN_b then go to step 4c.
If AC_a not equal/equivalent to AC_b then go to step 4b.
Conclude that fusion is possible, notify user (domain expert), suggest refinement action(s) and exit.

Step 4b: Check for fusion (continued):
If C_a not equal/equivalent to C_b then exit.
Conclude that fusion is possible, notify user (domain expert), suggest refinement action(s) and exit.

Step 4c: Check for fusion (continued):
If AC_a not equal/equivalent to AC_b then exit.
If C_a not equal/equivalent to C_b then exit.
If AN_a can be fused with AN_b [using Rule 4 (a)] then conclude that fusion is possible, notify user (domain expert), suggest refinement action(s) and exit.

4. IMPLEMENTATION

ConEditor+ is implemented in the Java programming language. The domain ontology in the Web Ontology Language (OWL; McGuinness & Harmelen, 2004) was developed using Protégé (Noy et al., 2000) and parsed using Jena (Seaborne, 2004). Figure 4 shows the domain ontology developed for the kite domain using the Protégé editor. ConEditor+ converts the ontology in OWL into an equivalent P/FDM Daplex schema using a transformation program developed in Java. This conversion is currently required as we have used an already existing constraint language (CoLan) that was developed for databases (Bassiliades & Gray, 1995; Gray et al., 2001). A transformation program to convert a XML DTD specification into Daplex schema has been implemented previously in Selpi (2004). The Daplex schema is used by the Daplex compiler within ConEditor+ to compile constraints in CoLan and detect any syntactic errors. The Daplex Schema is also used by a translator developed in Prolog to convert the constraints in CoLan into a semantic web[4] enabled XML constraint interchange format (CIF; Gray et al., 2001). ConEditor+ uses this machine-interpretable format (CIF) to detect inconsistencies (contradictions) and to suggest various ways to refine (fuse constraints, eliminate redundancies and subsumptions) the knowledge base prior to constraint solving. ConEditor+ performs a static comparison of all possible pairs of constraint expressions, that is, ConEditor+ compares constraints at the syntactical level, rather than comparing the solution sets. Thus, ConEditor+ is comparing pairs of constraints of the form, for example, P(x1, x2) and Q(x1, x3, a) and P(x1, x2) and Q(x1, x3, b), and by looking at the values of the constants (a, b), and the structure of the predicates (P, Q), working out that there is an inconsistency, subsumption, redundancy, or fusion. Comparison of all possible pairs of constraints results in time complexity of $O(n^2)$. Further, in each comparison, all the terms in one constraint are compared with all the corresponding terms in another constraint. Hence, the complexity of each comparison is $O(n^2)$. Comparison of all possible pairs of constraints is sufficient (or complete) for detecting redundancy and subsumption.

[4] The semantic Web is an evolving extension of the World Wide Web in which Web content can be expressed in a form that can be understood, interpreted, and used by computers to find, share, and integrate information more easily (Berners-Lee et al., 2001).

4.1. Redundancy

Consider n constraints, namely, $S_1, S_2, \ldots, S_n$. Let us assume $S_1 \equiv S_2 \equiv \cdots \equiv S_n$, that is, redundancy exists between all the n constraints. By comparing all possible pairs of constraints, ConEditor+ detects the following ${}^nC_2$ cases: $S_1 \equiv S_2$, $S_1 \equiv S_3, \ldots, S_1 \equiv S_n, S_2 \equiv S_3, \ldots, S_2 \equiv S_n, \ldots, S_{n-1} \equiv S_n$. One can infer from the above ${}^nC_2$ cases that redundancy exists between all n constraints. Moreover, when the domain expert eliminates redundancy in each of the ${}^nC_2$ cases, redundancy between all the n constraints are eliminated.

4.2. Subsumption

The principles described in Section 4.1 also apply in 4.2. Consider n constraints, namely, $S_1, S_2, \ldots, S_n$. Let us assume S_1 subsumes $\{S_2, S_3, \ldots, S_n\}$, that is, one constraint subsumes all the other $n - 1$ constraints ($n > 2$). By comparing all possible pairs of constraints, ConEditor+ detects the following ${}^nC_2$ cases: S_1 subsumes S_2, S_1 subsumes $S_3, \ldots, S_1$ subsumes S_n. One can infer from the above ${}^nC_2$ cases that S_1 subsumes $\{S_2, S_3, \ldots, S_n\}$. Moreover, when the domain expert eliminates subsumption in each of the ${}^nC_2$ cases, all cases of subsumption are eliminated.

However, comparison of all possible pairs of constraints is insufficient (or incomplete) for detecting Inconsistency and Fusion.

4.3. Inconsistency

Consider n constraints, namely, $S_1, S_2, \ldots, S_n$. Let us assume $\forall x \{S_1: P(x) < Q(x), S_2: Q(x) < R(x), \ldots, S_n: R(x) < P(x)\}$, where $x \in C$, C is a class in the domain ontology, Q and R are properties in the domain ontology. By comparing S_1, S_2, and S_n, one can infer that there exists an inconsistency between them. This kind of inconsistency cannot be detected by comparing all pairs of constraints.

4.4. Fusion

Consider n constraints, namely, $S_1, S_2, \ldots, S_n$. Let us assume $S_1, S_2, \ldots, S_n$ could be fused into a single constraint S by applying the rule of fusion via class to n constraints, where $n > 2$. This kind of fusion cannot be detected by comparing all pairs of constraints.

The reasons/justification for comparing only pairs of constraints in ConEditor+ are as follows:

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

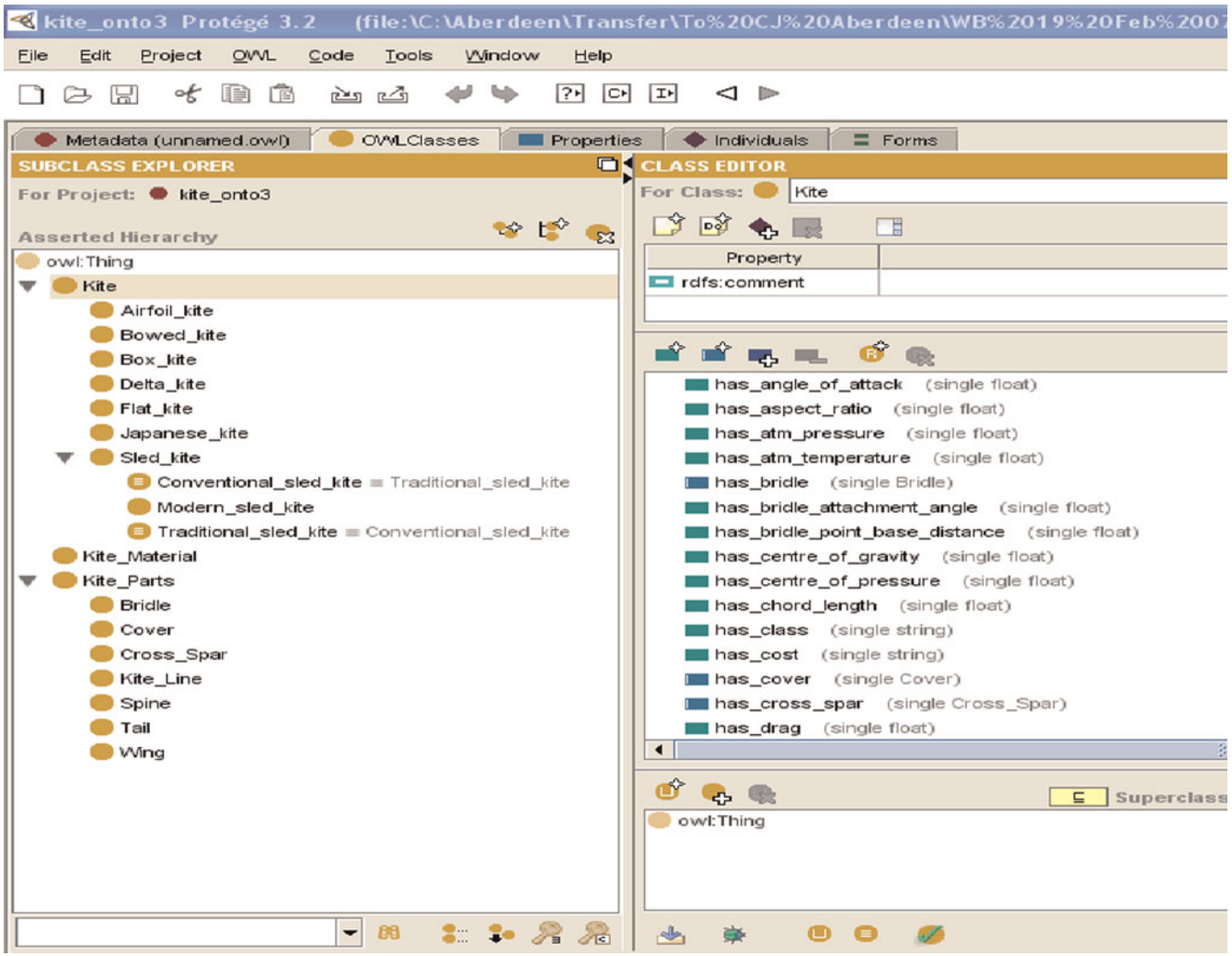

Fig. 4. The domain ontology of kites developed in Protégé. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

1. Comparison of all constraints (more than pairs) is more complex and substantially increases the complexity of the algorithm, especially when we consider an arbitrary number of first-order logic expressions. We plan to investigate this issue as part of the future work.
2. Moreover, the main aim of our research work is to demonstrate the usefulness of an explicit (machine-interpretable format) representation of design rationales (application conditions) in supporting the maintenance of constraints. The research aims and hypothesis have been specified in Section 1.2 in more detail. Details of the experiments conducted to evaluate our research work are provided in the following section.

5. EVALUATION

This section is divided into three parts: Section 5.1 describes a preliminary evaluation done at Rolls-Royce, Derby; Section 5.2 describes two experiments: Experiment 1 to address the main research question of our work and Experiment 2 to test the usability of ConEditor+; Section 5.3 describes an evaluation done to strengthen our claims by applying our proposed approach to an additional domain, consisting of a more demanding KB.

5.1. Preliminary evaluation

We performed a *preliminary* evaluation of ConEditor+ at Rolls-Royce, Derby. The main aim of this experiment was to determine whether the designers at Rolls-Royce would consider using ConEditor+ to capture design rules as constraints. A demonstration by one of the investigators (Suraj Ajit) was given to a group of five design engineers at Rolls-Royce. The demonstration involved the following phases:

Phase 1: Presenting the constraint as in the rule book, that is, as a mixture of textual and graphical information

The English rendering of the constraint is

Bolted joints must conform to the formula

$$N_{\min} = \text{PCD} + 2 * M + \text{Max. Nut Width}$$

where $N_{\min}$ is the trap diameter of the flange, PCD is the pitch circle diameter of the flange, $150.0 < \text{PCD} \leq 180.0$, and $M =$ gap in the flange $= 0.5$.

Phase 2: Expressing the constraint in CoLan

This constraint was expressed in CoLan by the investigator and discussed with the Rolls-Royce designers:

constrain each j **in** BoltedJoint
such that has_nut_type(j) = "Captive Nut" **and**
dimension(pitch_circle_diameter(has_flange(j))) ≥ 150.0
and dimension(pitch_circle_diameter(has_flange(j))) < 180.0
and is_internal(has_flange(j))
to have gap(has_flange(j)) = 0.5
and dimension(trap_diameter(has_flange(j)))
= dimension(pitch_circle_diameter(has_flange(j)))
+ 2 ∗ gap(has_flange(j))+dimension(nut_width(has_nut(j)))
+ tolerance(nut_width(has_nut(j)))

Phase 3: Formulating the constraint using ConEditor+

In the final stage of the demonstration, the CoLan expression was input to ConEditor+ by the investigator together with a description of the usage of ConEditor+'s GUI.

The design engineers were then asked to comment on ConEditor+ and, in particular, whether they would consider using ConEditor+ to capture design rules. The design engineers reported that they found ConEditor+ simple, user-friendly and intuitive to use. However, they reported that they would need some training before they could actually perform phases 2 and 3 unsupported. They also made the general point that the company has a Design Standards group that has the responsibility for creating and maintaining the company-wide rule book(s). They would expect this group to use systems such as ConEditor+ to formulate constraints. The designers would then subsequently use the information either in the current form or in a Designers' Workbench-like environment.

5.2. Experiments

Following the preliminary evaluation, two experiments were conducted and the details of these experiments are given below.

5.2.1. Experiment 1

The aim of this experiment was to address the following research question: Could an explicit representation of application conditions together with the constraints and the domain ontology help a machine in: reducing the number of inconsistencies and detecting subsumption, redundancy, fusion, and suggesting appropriate refinements between pairs of constraints?

We studied the kite design domain and captured constraints together with the corresponding application conditions (rationales). We ran an experiment with ConEditor+ using: KB_1 containing 15 constraints together with their application conditions, KB_2 containing the same constraints without any application conditions. The reader is encouraged to refer Ajit (2008) for the complete list of constraints and the corresponding application conditions that have been captured from the kite design domain.

Results. For KB_1, ConEditor+ detected three subsumptions, zero contradictions, three redundancies, and two cases of fusion between pairs of constraints. For KB_2, ConEditor+ detected two subsumptions, five contradictions, three redundancies, and four cases of fusion between pairs of constraints. For KB_2, it is evident that the absence of application conditions caused a number of inconsistencies (five

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

contradictions), and also, ConEditor+ suggested a number of inappropriate refinements. This is explained further below.

For example, let us consider two KBs, namely, A and B, containing the following constraints:

KB A (with application conditions):

(i) **constrain each** k **in** Kite **such that** has_level(k) = "beginner" **to have** density(has_material(has_cover(k))) < 0.5

(ii) **constrain each** k **in** Kite **such that** has_level(k) = "advanced" **to have** density(has_material(has_cover(k))) > 1.0

KB B (without application conditions):

(iii) **constrain each** k **in** Kite **to have** density(has_material(has_cover(k))) < 0.5

(iv) **constrain each** k **in** Kite **to have** density(has_material(has_cover(k))) > 1.0

As shown above, the KB A contains two constraints, (i) and (ii), with the corresponding application conditions. The KB B contains the same pair of constraints, (iii) and (iv), without the corresponding application conditions. For KB A, ConEditor+ does not detect any inconsistency. For KB B, ConEditor+ detects a contradiction between the two constraints, (iii) and (iv). Hence, it can be concluded that the absence of application conditions could cause inconsistencies between constraints. In addition, this can cause ConEditor+ to suggest inappropriate refinements as shown below.

For example, let us consider two KBs, namely, C and D, containing the following constraints:

KB C (with application conditions):

(v) **constrain each** d **in** Delta_kite **such that** has_level(d) = "beginner"**to have** bridle_length(has_bridle(d)) > 3 ∗ has_height(d)

(vi) **constrain each** d **in** Delta_kite **such that** has_wind_condition(d) = "strong" **to have** kite_line_strength(has_kite_line(d)) > 90

KB D (without application conditions):

(vii) **constrain each** d **in** Delta_kite **to have** bridle_length(has_bridle(d)) > 3 ∗ has_height(d)

(viii) **constrain each** d **in** Delta_kite **to have** kite_line_strength(has_kite_line(d)) > 90

Again, we have considered two KBs C and D, with and without application conditions, respectively. For KB C, ConEditor+ does not suggest any refinement. For KB D, ConEditor+ inappropriately suggests that the two constraints, (vii) and (viii), be fused and replaced by the constraint (ix):

(ix) **constrain each** d **in** Delta_kite **to have** bridle_length (has_bridle(d)) > 3 ∗ has_height(d) and kite_line_strength(has_kite_line(d)) > 90

Conclusion. One can infer from the results of Experiment 1 and the examples described above that an explicit representation of the application conditions together with the constraint reduced the number of inconsistencies and also prevented ConEditor+ from suggesting inappropriate refinements.

5.2.2. Experiment 2

The aim of this experiment was to determine the usability of ConEditor+. In particular, we aimed to seek answers for the following main questions (Rubin, 1994; Dumas & Redish, 1999; Barnum, 2002):

a. Could the subjects successfully perform the allocated tasks within the time benchmark?
b. Did the subjects perform the tasks accurately? What kind of mistakes did the subjects make (if any)? Could the GUI be modified to eliminate or minimize these errors?
c. How easy and intuitive did the subjects find the system to use?

A demonstration was given by the developer of ConEditor+ to each of the five subjects (two mechanical engineering research students, two computer science research students, and one computer science research fellow) individually. The demonstration was given using instructions from a script to maintain consistency and consisted of the following main tasks: description of all the features of ConEditor+; a walkthrough of the process of converting a sample constraint in English to CoLan, inputting the CoLan constraint using ConEditor+, eliminating syntactic errors and performing appropriate refinements (redundancy, subsumption, contradiction, fusion). Each subject was then asked to perform the following tasks.

Task 1: The following constraint was presented in English and CoLan.

English: "Every standard sized or stunt type Sled Kite must have a kite line with strength greater than or equal to 15 units."

CoLan:

constrain each s **in** SledKite **such that** has_size(c) ="standard" or has_type(s) = "stunt" **to have** kites_line_strength(has_kite_line(c)) >= 15

The subject was asked to input the above constraint in CoLan using ConEditor+.

Task 2: ConEditor+ already consisted of a constraint (shown below) in its KB that was subsumed by the constraint the subject input in Task 1. After successfully inputting the constraint in Task 1, ConEditor+ detects subsumption and suggests the user to consider deleting the following constraint:

constrain each c **in** ConventionalSledKite **such that** has_size(c) ="standard" **to have** kites_line_strength (has_kite_line(c)) >= 15

Each subject was asked to follow ConEditor+'s suggestion and delete the above constraint.

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

Task 3: Each subject was asked to answer a questionnaire and also provide oral feedback on the usability of ConEditor+ to its developer. The questionnaire contained various questions regarding the usability and usefulness of various features of ConEditor+. The subjects were asked to use a five-point rating scale (1 = poor, 5 = excellent). More details about the experiment and the questionnaire used can be found in Ajit (2008). The developer observed all the actions performed by each subject and took notes. A pilot experiment was conducted before the actual experiment using a computer science research student as the subject and that helped rectify some elementary errors in the script and GUI.

Results. All the subjects ultimately completed the allocated tasks accurately within the corresponding time benchmarks. Tasks 1 and 2 were allocated a time benchmark of 5 and 3 min each, respectively. The subjects were not aware of this time benchmark. All the errors committed by subjects can be summarized as follows: two subjects double clicked on the keywords panel instead of a single click. This resulted in the selected keyword being appended twice to the constraint expression. The GUI has now been changed to support a double mouse click instead of a single click. Two subjects reported that they would like to see the console tab in the display panel activated automatically after inputting a constraint rather than manually activating the console tab. The GUI was modified to support this feature. Two subjects also suggested that they would like a search facility being provided in the taxonomy panel to be able to easily locate entities in a large taxonomy. We plan to incorporate this feature as part of the future work. All the subjects reported that they found ConEditor+ easy to use and helpful for the maintenance of constraints. The overall rating given by the subjects, for the usability (including capture and maintenance facilities) of ConEditor+ was 3.8 (see graph in Figure 5). Thus, the results of Experiment 2 indicate that ConEditor+ is easy to use and it aids the capture and maintenance of constraints.

5.3. Extension/evaluation of jet engine ontology and maintenance of a more complex set of constraints

After successful application and evaluation of ConEditor+ in the domain of kite design, we decided to apply our proposed approach to part of the considerably more demanding Rolls-Royce domain. We initially reviewed the ontology used to support Designers' Workbench, and then analyzed a considerable number of additional Rolls-Royce's design standard documents (72) that contain rules/standards for the design of various parts and processes involved in civil aeroengines. Interviews were held with a design engineer at Rolls-Royce, Derby. We then extended the jet engine ontology to incorporate the additional information (e.g., classes, properties) obtained from these analyses. The jet engine ontology was then evaluated by a domain expert in Rolls-Royce. Following several discussions with the domain expert and modifications to the ontology, the ontology was approved by the domain expert. A confidential technical report (Ajit et al., 2007) describes the list of all constraints and application conditions obtained from the analysis of design rule books for part of the Rolls-Royce domain, together with their corresponding representations in CoLan.

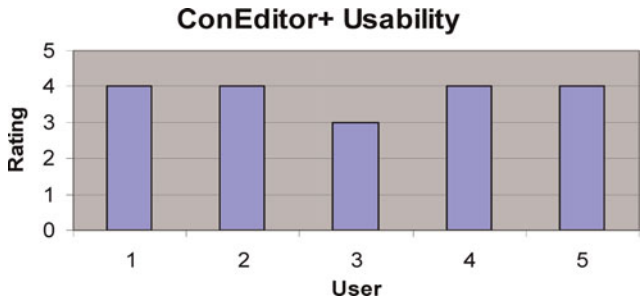

Fig. 5. A graph showing the results of an experiment to evaluate the usability of ConEditor+. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

6. RELEVANT WORK

In product development and design, constraints arise in many forms. Constraints can either be represented as rules or objects (Sriram & Maher, 1986). One of the first attempts to manage constraints for automation of computation in engineering applications was the work done by Harary (1962) and Steward (1962). Since then, there has been considerable amount of work done on the representation, use, and management of constraints including the development of rule-based systems (Frayman & Mittal, 1987; Wielinga & Schreiber, 1997; Junker, 2001), and also in the field of diagnosis (Felfernig et al., 2004). Constraint management done in systems above mainly refers to the *detection* of redundant and contradictory constraints during *constraint solving*, whereas ConEditor+ detects redundant, subsumed, contradictory, and fusible constraints prior to constraint solving. ConEditor+ compares pairs of constraints by looking at the values of the constants, and the structure of the predicates rather than by computing the solution sets of constraints.

It became important to represent the defaults and preferences declaratively as constraints, rather than encoding them in the procedural parts of the program (Borning et al., 1989). In most cases, domain-oriented or method-oriented tools (in the form of templates) were provided to capture constraints/rules from the domain experts. The cost of developing such tools is high, especially when their restricted scope is taken into account (Eriksson et al., 1995). In comparison to the above tools, ConEditor+ is a domain-independent tool that can be used by domain experts to capture constraints using the appropriate domain ontology. These constraints are converted into a standard format (in CIF) for use by other systems. A similar tool for capturing constraints has been developed by Gray and Kemp (2006). This tool uses a diagrammatic representation in the form of a relationship graph to

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

capture constraints. The drawback of this tool is that the diagram can become cumbersome for large domain ontologies.

There has been a lot of work done on the verification of knowledge-based systems. Suwa et al. (1982) are credited with one of the earliest works in automated verification. Other notable works include Nguyen et al. (1985), Preece et al. (1992), Zlatareva (1998), Hicks (2003), and Qian et al. (2005). In comparison to the above-cited work, the focus of our research is to prevent errors in the KB as much as possible. We believe that it is important to explicitly represent the assumptions and contexts in which each constraint is applicable, and that this would prevent a substantial number of errors from occurring in the KB. Subsequently, we use the proposed knowledge refinement rules to detect errors (or anomalies), and also suggest ways to refine (simplify/optimize) the KB.

CoLan is close to the constraint language Galileo proposed by Bowen et al. (1990) that has been used to support conceptual design and design knowledge representation. Both CoLan and Galileo are based on first-order logic, and can be used to express both existentially and universally quantified constraints. However, we believe CoLan provides better readability for domain experts compared to Galileo and other constraint programming languages such as ILOG OPL language (Junker & Mailharro, 2003). Moreover, CoLan was developed by one of our colleagues, and we have the software to convert Colan into standard XML CIF format that makes it portable. Also, Colan is mainly used in ConEditor+ as a declarative language for expressing constraints and not used for constraint programming. CoLan is converted into CIF, which in turn, is converted into a query in RDQL and a predicate in Prolog by the Designers' Workbench for constraint processing.

Design rationale systems capture a lot more information regarding the reasoning of design decisions. However, design rationale systems (Regli et al., 2000; Bracewell & Wallace, 2003) usually capture the information in a human readable format. Although the information may have some structure, the information cannot be understood, interpreted, and used by machines to provide benefits to the designers immediately. Design rationales are difficult to retrieve, and hence rarely used. ConEditor+ captures application conditions as rationales together with the constraints and uses the information (including domain ontology) to detect inconsistency, subsumption, redundancy, fusion, and suggest appropriate refinements between pairs of constraints to designers. This should encourage designers to input application conditions associated with the constraints because it provides immediate benefits.

7. CONCLUSIONS AND FUTURE WORK

This article describes a methodology together with a system that has been developed to enable domain experts to capture and maintain constraints in an engineering design environment. The context is a system known as the Designers' Workbench, developed to support engineering designers by checking that their configurations satisfy all the constraints. The Designers' Workbench is faced with the task of accumulating the constraints. This requires a knowledge engineer to study the design rule book(s), consult the design engineer (domain expert), and encode all the constraints into the Designers' Workbench's KB. This is a tedious, time-consuming, and error-prone task. Hence, we have developed a system, ConEditor+, to enable domain experts themselves to capture and maintain engineering design constraints.

We believe that to apply constraints appropriately, it is necessary to capture the contexts and assumptions associated with constraints and an explicit representation of this information (rationales) referred to as application conditions would be extremely beneficial to both humans and machines to support the maintenance of constraints. We have proposed four main types of knowledge refinement rules that use the application conditions together with the constraints and the domain ontology to detect inconsistencies, subsumption, redundancy, and fusion. We implemented these rules in ConEditor+, and demonstrated with the help of an experiment that an explicit representation of application conditions together with the constraints and the domain ontology could help the machine in reducing the number of inconsistencies and detecting subsumption, redundancy, fusion, and suggesting appropriate refinements between pairs of constraints. We also believe that ConEditor+ is a useful tool for domain experts to capture and maintain constraints. The evaluation of ConEditor+'s usability has given us encouraging results. Further, we applied our proposed methodology and tool to part of the more demanding Rolls-Royce domain to strengthen our claims.

The proposed architecture (Fig. 6) shows how ConEditor+ fits into a much broader framework. A Design Standards author initially inputs all the design rules (constraints) in CoLan together with the associated application conditions into ConEditor+. The design constraints and application conditions are then converted from CoLan into CIF. CIF is further converted into Prolog predicates and RDQL queries and processed by the Designers' Workbench. ConEditor+ uses the constraints and application conditions represented in CIF together with the domain ontology in OWL to detect inconsistencies, subsumption, redundancy, fusion, and suggest appropriate refinements between pairs of constraints to support maintenance. It is planned to interface the Designers' Workbench to a more sophisticated CAD/KBE system as part of the future work. We also have plans to make ConEditor+ into a Protege (Noy et al., 2000) plug-in that would involve converting the constraints into CIF/SWRL (McKenzie et al., 2004).

ACKNOWLEDGMENTS

We acknowledge the financial support provided by the EPSRC Sponsored Advanced Knowledge Technologies project, GR/N15764, which is an Interdisciplinary Research Collaboration involving the University of Aberdeen, the University of Edinburgh,

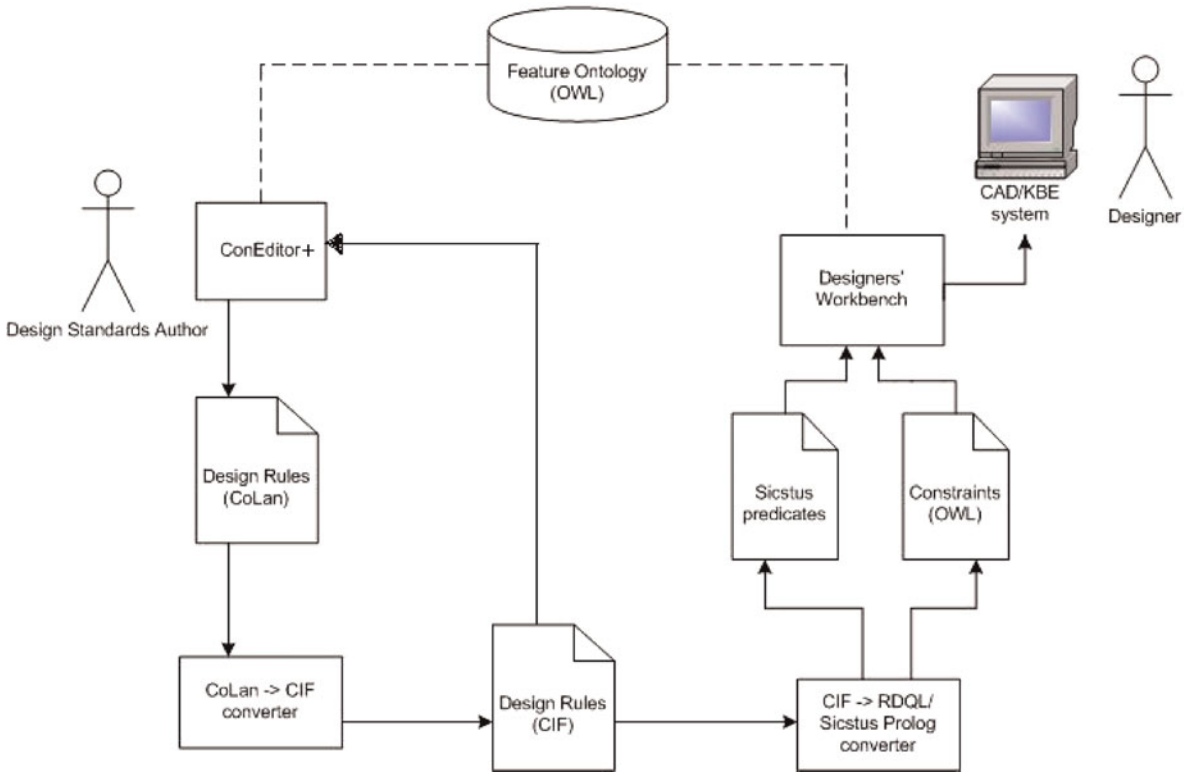

Fig. 6. The proposed system architecture. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

the Open University, the University of Sheffield, and the University of Southampton. We also acknowledge the substantial contributions of Mr. Stephen Docherty from the Transmissions and Structures division and Mr. Colin Cadas of Rolls-Royce plc, Derby, UK. We are extremely grateful to Dr. Kit Hui and Professor Peter Gray for providing us with the software to convert constraints in CoLan to CIF.

REFERENCES

Ajit, S. (2008). *Capture and maintenance of constraints in engineering design*. PhD thesis, Department of Computing Science, University of Aberdeen, Aberdeen, UK.

Ajit, S., Sleeman, D., Fowler, D.W., Knott, D., & Hui, K. (2005). Acquisition and maintenance of constraints in engineering design. *Proc. 3rd Int. Conf. Knowledge Capture, KCAP 2005*, pp. 173–174, Banff, Canada.

Ajit, S., Sleeman, D., & Knott, D. (2007). *Analysis of Design Rule Books of Part of the Rolls-Royce Domain*, Technical Report. Department of Computing Science, University of Aberdeen, Aberdeen, UK.

AKA. (2006). American Kite Association. Accessed at http://www.aka.org.au/kites_in_the_classroom/index.htm on June 28, 2006.

Bahler, D., & Bowen, J. (1992). Design rationale management in concurrent engineering. *Workshop on Design Rationale Capture and Use, 10th National Conf. Artificial Intelligence (AAAI-92)*, San Jose, CA.

Barker, V.E., & O'Connor, D.E. (1989). Expert systems for configuration at digital: XCON and beyond. *Communications of the ACM 32(3)*, 298–318.

Barnum, C.M. (2002). *Usability Testing and Research*. Upper Saddle River, NJ: Allyn & Bacon.

Bassiliades, N., & Gray, P. (1995). CoLan: a functional constraint language and its implementation. *Data & Knowledge Engineering 14(3)*, 203–249.

Borning, A., Maher, M., Martindale, A., & Wilson, M. (1989). Constraint hierarchies and logic programming. *Int. Conf. Logic Programming (ICLP)*, PP. 149–164, Lisbon, Portugal.

Bowen, J., O'Grady, P., & Smith, L. (1990). A constraint programming language for life-cycle engineering. *Artificial Intelligence in Engineering 5(4)*, 206–220.

Bracewell, R.H., & Wallace, K.M. (2003). A tool for capturing design rationale. Proc. *Int. Conf. Engineering Design (ICED 03)*, Stockholm.

Brown, D.C. (2006). Assumptions in design and design rationale. *Design Rationale Workshop, DCC'06*, Eindhoven, The Netherlands.

Bultman, A., Kuipers, J., & Harmelen, F.V. (2000). Maintenance of KBS's by domain experts: the Holy Grail in practice. *Thirteenth Int. Conf. Industrial & Engineering Applications of Artificial Intelligence & Expert Systems IEA/AIE'00*.

Burge, J., & Brown, D.C. (2003). Rationale support for maintenance of large scale systems. *Workshop on Evolution of Large-Scale Industrial Software Applications (ELISA), ICSM '03*, Amsterdam.

CEKS. (2006). *Cutting Edge Kite Shop*. Accessed at http://www.cuttingedgekites.com/faq.htm on June 28, 2006.

Coenen, F.P. (1992). A methodology for the maintenance of knowledge based systems. *EXPERSYS-92 (Proc.), IITT-Int.* (Niku-Lari, A., Ed.), pp. 171–176.

Dumas, J.S., & Redish, J.C. (1999). *A Practical Guide to Usability Testing*. Bristol: Intellect Books.

Eden, M. (1998). *The Magnificient Book of Kites: Explorations in Design, Construction, Enjoyment and Flight*. New York: Black Dog & Levanthal Publishers.

Eriksson, H., Puerta, A., Gennari, J., Rothenfluh, T., Tu, S., & Musen, M. (1995). Custom-tailored development tools for knowledge-based systems. *Proc. Ninth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada.

Felfernig, A., Friedrich, G., Jannach, D., & Stumptner, M. (2004). Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence 152*, 213–234.

Fletcher, D., & Gu, P. (2005). Adaptable design for design reuse. *Second CDEN Int. Conf. Design Education, Innovation, and Practice*.

Fowler, D.W., Sleeman, D., Wills, G., Lyon, T., & Knott, D. (2004). Designers' Workbench. *Proc. 24th SGAI Int. Conf. Innovative Techniques and Applications of Artificial Intelligence*, pp. 209–221, Cambridge.

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

Frayman, F., & Mittal, S. (1987). COSSACK: a constraints-based expert system for configuration tasks. *Knowledge Based Expert Systems in Engineering: Planning and Design* (Sriram, D., & Adey, R.A., Eds.), pp. 143–166. Southampton: Computational Mechanics Publications.
Goonetillake, J.S., & Wikramanayake, G.N. (2004). Management of evolving constraints in a computerised engineering design environment. *Proc. 23rd National IT Conf.*, Colombo, Sri Lanka.
Gray, P., Hui, K., & Preece, A. (2001). An expressive constraint language for semantic web applications. *E-Business and the Intelligent Web: Papers From the IJCAI-01 Workshop*, pp. 46–53, Seattle, WA.
Gray, P., & Kemp, G. (2006). Capturing quantified constraints in FOL, through interaction with a relationship graph. *15th Int. Conf. Knowledge Engineering and Knowledge Management (EKAW 2006)*, Podebrady, Czech Republic.
Gruber, T.R. (1995). Towards principles for the design of ontologies used for for knowledge sharing. *International Journal of Human–Computer Studies 43(5–6)*, 907–928.
Grudin, J. (1996). Evaluating opportunities for design rationale capture. In *Design Rationale: Concepts, Techniques, and Use* (Carroll, J.M., Ed.). Mahwah, NJ: Erlbaum.
Harary, F. (1962). A graph theoretic approach to matrix inversion by partitioning. *Numerische Mathematik 4*, 128–135.
Hicks, R.C. (2003). Knowledge base management systems—tools for creating verified intelligent systems. *Knowledge-Based Systems 16*, 165–171.
Hooey, B.L., & Foyle, D.C. (2007). Requirements for a design rationale capture tool to support NASA's complex systems. In *Int. Workshop on Managing Knowledge for Space Missions*, Pasadena, CA.
Junker, U. (2001). Quickxplain: conflict detection for arbitrary constraint propagation algorithms. *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints (CONS-1)*, Seattle, WA.
Junker, U., & Mailharro, D. (2003). The logic of ilog(j) configurator: combining constraint programming with a description logic. *Proc. IJCAI'03 Workshop on Configuration*, Acapulco, Mexico.
Leigh, D. (2006). *Delta kite designs.* Accessed at http://www.deltas.freeserve.co.uk/home.html on June 28, 2006.
Lin, L., & Chen, L.C. (2002). Constraints modelling in product design. *Journal of Engineering Design 13(3)*, 205–214.
Lords, D. (2006). *Kite, kite buggy and land yacht page.* Accessed at http://users.techline.com/lord/index.html on June 28, 2006.
McGuinness, D.L., & Harmelen, F.v. (2004). *OWL Web Ontology Language overview, W3C recommendation February 10, 2004.* Accessed at http://www.w3.org/TR/owl-features/ on August 29, 2006.
McKenzie, C., Gray, P., & Preece, A. (2004). Extending SWRL to express fully-quantified constraints. *Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004), Int. Semantic Web Conf.*, pp. 139–154, Hiroshima, Japan.
McMahon, C., Lowe, A., & Culley, S. (2004). Knowledge management in engineering design: personalization and codification. *Journal of Engineering Design 15(4)*, 307–325.
Meseguer, P., & Preece, A.D. (1995). Verification and validation of knowledge-based systems with formal specifications. *Knowledge Engineering Review 10*, 331–343.
Nguyen, T.A., Perkins, W.A., Laffey, T.J., & Pecora, D. (1985). Checking an expert systems knowledge base for consistency and completeness. *IJCAI '85*, Vol. 1, pp. 375–378, Los Angeles.
Noy, N.F., Fergerson, R.W., & Musen, M.A. (2000). The knowledge model of Protege-2000: combining interoperability and flexibility. *Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW 2000)*, Juan-les-Pins, France.
Preece, A.D., Shinghal, R., & Batarekh, A. (1992). Verifying expert systems: a logical framework and a practical tool. *Expert Systems with Applications 5(3/4)*, 421–436.
Qian, Y., Zheng, M., Li, X., & Lin, L. (2005). Implementation of knowledge maintenance modules in an expert system for fault diagnosis of chemical process operation. *Expert Systems with Applications 28*, 249–257.
Regli, W.C., Hu, X., Atwood, M., & Sun, W. (2000). A survey of design rationale systems: approaches, representation, capture and retrieval. *Engineering with Computers: An International Journal for Simulation-Based Engineering 16*, 209–235.
Rubin, J. (1994). *Handbook of Usability Testing*. New York: Wiley Technical Communication Library.
Seaborne, A. (2004). *RDQL—a query language for RDF.* Accessed at http://www.w3.org/Submission/RDQL/ on August 29, 2006.
Selpi. (2004). *An FDM prototype for pathway and protein interaction data*. Master's Thesis, Chalmers University of Technology, Goteborg, Sweden.
Serrano, D., & Gossard, D. (1992). Tools and techniques for conceptual design. In *Artificial Intelligence in Engineering Design* (Tong, C. & Sriram, D., Eds.), Vol. 1, pp. 71–116. San Diego, CA: Academic.
Soloway, E., Bachant, J., & Jensen, K. (1987). Assessing the maintainability of XCON-in-RIME: coping with problems of a very large rule-base. In *Proc. AAAI-87*, pp. 824–829, Seattle, WA.
Sriram, D., & Maher, M.L. (1986). The representation and use of constraints in structural design. In *Applications of Artificial Intelligence in Engineering Problems*, Vol. 1, pp. 355–368. Southampton: Computational Mechanics Publications.
Steward, D.V. (1962). On an approach to techniques for the analysis of the structure of large systems of equations. *SIAM Review 4*.
Streeter, T. (1980). *The Art of the Japanese Kite*. Tokyo: Charles E. Tuttle Company.
Suwa, M., Scott, A.C., & Shortliffe, E.H. (1982). An approach to verifying completeness and consistency in a rule-based system. *AI Magazine 3(4)*, 16–21.
Wardley, A. (2006). *Basics of stunt kite design*. Accessed at http://www.kfs.org/~abw/kite/rec.kites/skdesign1.html on June 28, 2006.
Wielinga, B., & Schreiber, G. (1997). Configuration-design problem solving. *IEEE Expert 12(2)*, 49–57.
Yolen, W. (1976). *The Complete Book of Kites and Kite Flying*. New York: Simon and Schuster Trade.
Zlatareva, N.P. (1998). A refinement framework to support validation and maintenance of knowledge-based systems. *Expert Systems with Applications 15*, 245–252.

Suraj Ajit is a doctoral student in the Department of Computing Science at the University of Aberdeen. He is currently employed by the University of Dundee to work with Calico Jack Ltd. on industrial applications of ontologies. From 2002 to 2006 he worked as a Research Assistant for the Advanced Knowledge Technologies Project. He received a BE (first class) degree in computer science from Bangalore University in 2001 and will graduate with a PhD in computing science from the University of Aberdeen in 2008. Suraj's main research interests are in knowledge management, constraints, engineering design, and ontologies.

Derek Sleeman is a Professor of Computing Science at the University of Aberdeen. He was one of the Principal Investigators of the EPSRC-sponsored IRC in Advanced Knowledge Technologies (2000–2007). Derek's research activities have remained at the intersection of artificial intelligence and cognitive science, but his focus has moved from intelligent tutoring systems to cooperative knowledge acquisition, knowledge refinement systems, and KB reuse. He was a program committee member for the International, European, and National conferences in machine learning and knowledge acquisition and capture and Conference Chair for K-CAP 2007. Dr. Sleeman also served on various editorial boards, including the *Machine Learning Journal* and the *International Journal of Human–Computer Sciences.* He was made a Fellow of the European AI Societies in 2004.

David W. Fowler is currently a Research Fellow at the University of Aberdeen. He received his BSc in computer science from Heriot–Watt University, Edinburgh, and his MSc in artificial intelligence and automated reasoning from Queen Mary College, University of London. David received his PhD from the University of Aberdeen in 2002. Dr. Fowler's

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press

main research interests are in knowledge representation and ontologies applied to engineering domains and in constraint satisfaction under uncertainty.

David Knott joined Rolls-Royce in 1977 as an undergraduate apprentice. He graduated from Loughborough University in 1981 with a first class degree in mechanical engineering. In 2000 he was appointed Company Specialist—Design Technology, with responsibility for improving the design process across Rolls-Royce by acquiring appropriate technology and supporting its application to the company's products and processes. He is currently leading two DTI funded research projects involving multidisciplinary academic and cross-sector industrial collaboration. David is a Chartered Mechanical Engineer and Fellow of the Institute of Mechanical Engineers.

https://doi.org/10.1017/S089006040800022X Published online by Cambridge University Press