*Artificial Intelligence for Engineering Design, Analysis and Manufacturing* (1998), *12*, 333–344. Printed in the USA.
Copyright © 1998 Cambridge University Press 0890-0604/98 $12.50

# Conceptual modelling for configuration: A description logic-based approach

DEBORAH L. MCGUINNESS[1] AND JON R. WRIGHT
AT&T Labs–Research, A215, 180 Park Avenue, Florham Park, NJ 07932, U.S.A.

(Received November 5, 1997; Accepted February 5, 1998)

**Abstract**

Representing objects and their interactions can be quite challenging when an application requires many complicated, interconnected objects that are restricted in how they can be instantiated. In this paper, we present our approach to conceptual modelling. We have used this approach with success in a number of applications, the largest of which is the PROSE family of configurators. PROSE was first deployed in 1990 and has been used to configure over 4 billion dollars worth of AT&T and Lucent telecommunications equipment. We will discuss our approach to conceptual modelling, which is based on knowledge representation, show how it meets our representation and reasoning needs, and then discuss the relative merits of the approach.[2]

**Keywords:** Conceptual Modelling; Configuration; Knowledge Representation; Description Logics

## 1. INTRODUCTION

Conceptual modelling is an outgrowth of interdisciplinary work in knowledge engineering, knowledge representation, programming languages, and data modelling over the past ten to fifteen years (Brodie et al., 1982; Loucopoulos & Zicari, 1992). We now have nearly a decade of experience both developing and deploying configurator applications based on conceptual models implemented in description logic (see Wright et al., 1993). There have been many different techniques and methods associated with configurator development. Interestingly enough, we think that the best arguments in favor of our approach are derived from sound software engineering practices and practical experience, especially in the context of small team programming projects.

Configurators usually evolve as small- to medium-scale projects. As such, they share many of the problems (as well as some of the advantages) associated with other software projects of similar scale. But in addition, configurator developers face special problems, or, at least, encounter certain problems more often than is typical of most software developers.

First, because it must represent the physical structure and sometimes even the logical structure of complex products, the output of a configurator is structured in very complex ways. It is usually nontrivial, for example, to verify that any particular output is correct by inspection, partly because substantial domain knowledge is required to do so. On many of our real-world, practical projects, for example, special experts were recruited by our development team to supply domain knowledge for testing, verification, and validation.

Second, configurator developers seldom have control of their delivery schedules. Rather, the life cycle of the configured product determines when a configurator must be delivered. Even after a configurator has been fully deployed, enhancements are driven by the pace of changes to the product, and not by the development team's sense of what can delivered when.

In the telecommunications domain, where we are most at home, product knowledge can change at a surprisingly rapid pace—we have measured changes as high as 50% per year in some unusual cases. Moreover, the fastest changing products are nearly always new introductions or top sellers—both strategically important classes of products. Hence, configurator maintenance and enhancement is at the same time both critical and fast-paced.

Reprint requests to: Deborah L. McGuinness, AT&T Labs–Research, A215, 180 Park Avenue, Florham Park, NJ 07932, U.S.A. E-mail: dlm@research.att.com.

[1]Current address: Computer Science Department, Gates Building 2A, Stanford University, Stanford, CA 94305. E-mail: dlm@ksl.stanford.edu

[2]This paper contains some material presented at the *AI in Distributed Information Networks Workshop of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.



https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

Third, configuration has become a source of interesting real-world problems for researchers interested in optimization. The practical motivation to generate provably lowest cost, highest performance, greatest price/performance ratio, etc., fits very nicely into an optimization framework. Configurators with the capability of producing output with such characteristics provide a distinct business advantage.

In our case, however, while still recognizing the importance of optimization, we have not focused on it much. Simply delivering consistent and correct configurations with respect to customer specifications has turned out to be an advance in the state of the art for large practical applications. In addition, we have sometimes observed, on a practical level, that the ability to generate *any* technically feasible solution can provide a distinct competitive advantage, regardless of whether the solution is optimized in any sense.

Software development is sometimes contrasted with the building of physical artifacts such as houses or bridges. Such physical construction projects impose an inherent but incomplete ordering on their subtask structure—a partial-order. For example, a house must have a foundation before walls can be built; walls need to be provided before a roof, etc. On the other hand, other subtasks can proceed in parallel, that is, installation of plumbing and wiring. Part of the secret of being a good contractor is having a good grasp of this subtask structure, knowing when to bring in the electricians, when the painters can start working, who can work effectively in parallel, etc.

Similarly, software development projects have a subtask structure. The typical subtask structure for a software project is more abstract than that associated with housing construction. In addition, software projects exhibit more variation in subtask structure from project to project. This is largely because the range of problems that can be addressed *via* software is inherently greater. Taking this into account, even experienced development teams often begin a new project with considerable uncertainty with respect to the true relationship among subtasks, and at times may even lack a sound conception of what the subtasks are. Because of the nature of software, this is not likely to change.

Maintaining consistency and compatibility among parallel subtasks as a software project progresses is a major challenge, and comes to dominate the decision-making processes of many projects. Many development teams elect not to take on certain tasks rather than risk upsetting the delicate interactions of cooperating modules. In general, developers can achieve local consistency by applying sound development practices such as code reviews, code inspections, and unit testing. But the most difficult problems to detect, understand, and solve are problems that cannot be isolated to specific lines or sections of source code. Programmers sometimes unknowingly make different assumptions about the required behavior of interacting components, and this always results in unexpected and incorrect behavior. It is also difficult to fully understand the consequences of making design changes in software systems where there are many interactions among components. As observed by Brooks (1975) so many years ago, programmers spend a great deal of their time coordinating their activities with those of other programmers. In many respects, the secret to programmer productivity is minimizing the burden of interaction for the front-line programmer.

In our view, such unshared assumptions are a major cause of failure in software projects, and grow more serious as projects become larger and involve more programmers. Even working on their own, programmers may find it difficult to understand the consequences of changing some piece of software they themselves wrote. And when the programmer and the original designer are different individuals (e.g., during software maintenance), uncovering the consequences of a particular design change is commonly accomplished by trial and error—a blind, time-consuming, low-level process.

Object-oriented techniques such as encapsulation and inheritance may help in such cases. In practice, however, refactoring of class designs is very common, often motivated by the programmer's growing understanding of a problem domain. Even for experienced developers, it can be quite difficult to foresee all the consequences of a particular design decision and even the best class designers are sometimes unable to predict how their classes will be used.

Making changes of any magnitude in a complex system is very intimidating since it is so very difficult to understand how all the components interact. Hence, over time many development teams adopt a conservative policy of agreeing to only the minimal changes necessary.

These are problems to which configurators are particularly susceptible. Not only is the output from a product configurator complex, but they also have particularly strict requirements with respect to consistency and correctness—because sellers of goods desire to produce accurate quotes that they can stand behind both in terms of functionality and price. In this context, a software defect that causes a configurator to misrepresent the cost or functionality of a multimillion dollar configuration is more critical than even optimization.

From this perspective, description logics have some very attractive features. We elected to design and implement our conceptual model for configuration in a description logic called the CLASSIC knowledge representation system (Borgida et al., 1989; Brachman et al., 1991; Weixelbaum, 1991). We will defend our choice of a knowledge representation-based approach in more detail after providing an introduction to description logics as exemplified by CLASSIC and after introducing our configuration tasks, as exemplified by PROSE (Product Offerings Expertise) (Wright et al., 1993; McGuinness & Wright, 1998). We also note that others have considered description logics for configuration applications (e.g., Owsnicke-Klewe, 1988; Rychtyckyj, 1996). We, however, have the longest application deployment history and the largest family of configurators. What will become evident is that the main reasons for our choice of CLASSIC are derived from our needs to have a tool with the following abilities:

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

1. Model an object-oriented domain.
2. Handle incrementally evolving specifications.
3. Support an extensible schema.
4. Provide active completion of knowledge.
5. Provide reasoning even when knowledge is incomplete.
6. Detect and maintain consistency.
7. Support retraction and truth maintenance.
8. Provide access to a declarative encoding of knowledge for maintenance and help desk support.

Our reasoning about this set of features is as follows. First, because maintenance and enhancement are such critical activities for configuration projects, features that support incremental development are essential (i.e., 2, 3, and 5). Second, declarative encoding of knowledge (8) in a natural way (1) is also critical. A somewhat unusual, but quite important, example of this is help-desk support. A help desk for configurators fields many challenges, from the sales team, from customers, even from members of the design and engineering community. It is quite legitimate for members of the user community to challenge why a certain kind of circuit pack is required, to understand the methods for calculating cable lengths, etc. The ability to reference a representation meaningful to end-users can resolve many such challenges. Next, retraction and truth maintenance (7) is a very valuable tool, both for developers and end-users. For example, this directly supports a very critical human activity—hypothetical reasoning, commonly termed "what-if." Essentially, retraction allows end-users to modify their input and observe the consequences of those modifications without having to recalculate an entire configuration. This allows customers to gain a thorough understanding of their options by directly exploring the problem space. Finally, and most importantly, the ability to provide active completion of knowledge in the form of deductive closure and to detect inconsistencies provides a distinct and critical advantage (2, 4, 6, and 7). As discussed, this directly supports individual programmers in understanding the consequences of changes and modifications to a complex body of code, and helps teams of cooperating programmers uncover inconsistent assumptions early on.

It is really this last feature that distinguishes description logics from other approaches such as production rules (McDermott, 1981), object-oriented methodology (Booch, 1996), and/or hybrid rule-based/object-oriented systems (Giarrantano & Riley, 1994). Description logics maintain a state of global consistency over a state of knowledge. It is not possible for one part of a configuration to become inconsistent with another.

Inconsistencies happen all too easily with production rule systems, especially when the application requires more than one programmer. Data is globally accessible and nothing prevents one rule overwriting the result of another rule. In addition, partly because there is no logical theory underlying a production system language, retraction cannot be properly supported. Hence, even if inconsistencies were detected, there is no way to role back the state of the system to the most recent consistent state.

In our view, this is the chief reason why production system applications are limited in scale, a position that is at least partly derived from our experience with deploying production system applications in telecommunications (Vesonder et al., 1983; Wright et al., 1988; Ackroff et al., 1990).

Modern object-oriented techniques, such as UML (Booch, 1996), attempt to achieve scale by careful design methodologies and information-hiding—by restricting and controlling the ways in which objects can interact with each other. Configuration problems, however, typically involve constraints that cross object boundaries. When the data about one object changes, then other objects (sometimes many other objects) may change and thus they must be checked for consistency. Under the what-if scenario, it may even be necessary to role back the current state of an object to some previous state. In other words, there is a tension between providing a data base that can be subjected to global consistency checking and existing object-oriented design methodologies.

Good object-oriented modelling systems provide checking of pre- and post-conditions during model development. Description logics, however, go a step further in that they perform an explicit calculation of the logical implications of pre- and post-conditions in object designs.

We believe complicated deductive tasks, such as configuration and provisioning, have a common set of needs. The list provided above is a starting point derived from practical experience with configurator development. We suggest that a solution meeting these needs can be leveraged. Our platform has handled the test of time well, producing 17 deployed configurators. Some have been deployed continuously since 1990 (although enhanced over time) and others have projected lifetimes that span into the next century.

In the rest of this paper, we will introduce our problem, supply an introduction to description logics using a simple configuration example, discuss the PROSE conceptual model, and describe the strengths and weaknesses of our approach. In particular, we want to emphasize the importance of model explanation in supporting the application during different phases of the software life cycle.

## 2. DESCRIPTION LOGICS AND CLASSIC

Description logics form a subfield of knowledge representation and are based on a formal logic. The field was motivated by the seminal work of Brachman, culminating in the KL-ONE knowledge representation system (Brachman & Schmolze, 1985). The field arose from the desire to give a precise meaning to the nodes and arcs in the widely used notation of semantic networks (Sowa, 1991). Semantic networks provide a compelling modelling basis since they allow knowledge engineers to create nodes for objects, to

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

provide structure in those objects by naming roles (possibly with associated restrictions), and to relate objects to each other by graphically connecting nodes with role-labeled arcs. The problems, as many pointed out, for example, (Woods, 1975; Brachman, 1979, 1983) had to do with the many unclear interpretations of just what was meant and implied by such a graphical notation. Description logics, earlier called terminological logics, structured inheritance networks, and KL-ONE-like systems, provided a formal syntax and semantics for this sort of network and thus formed the logical foundation on which deduction could be based. The description logic community coordinated to generate a common system specification for description logics called the knowledge representation system specification (KRSS) (Patil et al., 1992). This specification contains a description of the essential components of description logics. We present the syntax and semantics of a subset of this language for use in this paper in Table 1.

CLASSIC is a representative description logic with implementations in LISP, C, and C++. It is also one of the DL systems (MacGregor, 1991) most similar to the description logic KRSS, thus it is a reasonable representative choice. One of CLASSIC's design goals was to balance the tradeoff between expressivity and complexity while keeping in mind its primary goal of being manageable and efficient for real applications. In a sense, our application work provides an empirical test of the CLASSIC design goals.

### 2.1. A Simple conceptual model in CLASSIC

Discussing a simple model is probably the best way to introduce some terminology. Since there is as yet no widely accepted notation for discussing conceptual models, we will use some terms specific to our implementation language, CLASSIC. We will focus on a model for a simple, but fictitious, piece of equipment—an assembly for electronics gear with four slots for circuit packs. The model for this equipment is shown in Figure 1. The general idea is that incoming signals are received by the interface unit, shuttled to the switch packs where they are "switched", and sent back through the interface unit. An optional performance monitor is allowed to replace one of the switch packs. The design is not realistic but the ideas of interface, switching, power, and performance monitoring is pervasive in telecommunications equipment design. On the right are a few CLASSIC expressions that implement a model of the simple assembly. The top primitive concept, `SimpleAssembly`, has four roles—one for each type of circuit pack in the simple assembly—`power`, `switch`, `interface`, and `perf-monitor`. The expression also places some number restrictions on the roles that correspond to the number of circuit packs of each type that can be assigned to an assembly.

The second concept, `AssemblyWithSwitch`, in Figure 1 is a specialization of `SimpleAssembly`. It is a `SimpleAssembly` with an additional constraint—at least one switch pack. `AssemblyWithSwitch` inherits all of the information from its parent concept `SimpleAssembly`. `AssemblyWithSwitch` could also inherit information from other concepts *via* multiple inheritance if necessary.

All of the concept definitions in a description logic-based model form a generalization hierarchy. Any concept B, whether stated or derived by inference, that is strictly more specific than another concept A is said to be a subconcept of A. In other words, B is a subconcept of A if and only if it is impossible to be an instance of B with being an instance of A—that is, in every possible interpretation, an instance of B must be an instance of A. A is then said to subsume B.

Consider a concept of a `SimpleAssembly` with two or more switches, shown in the following CLASSIC code fragment.

```
(define-concept AssemblyWithTwoSwitches
  (and
  SimpleAssembly
  (at-least 2 switch)))
```

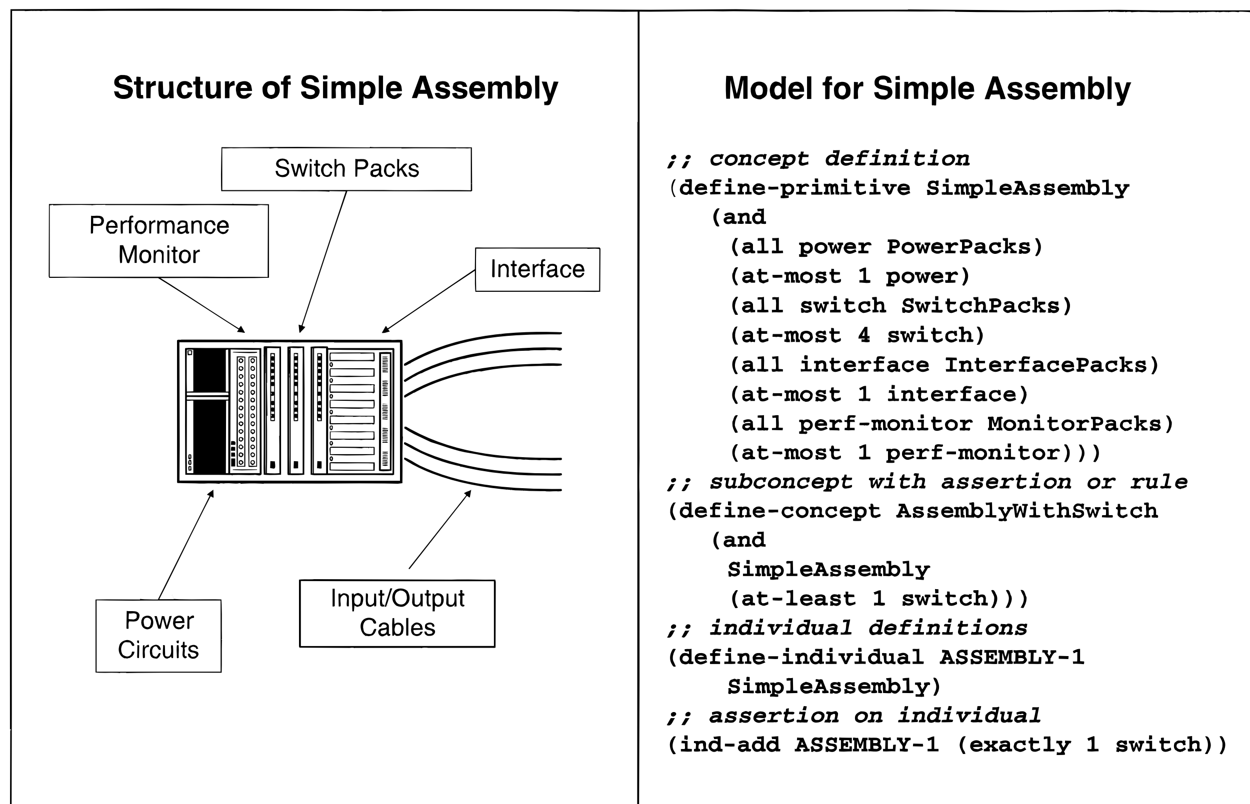
In this case, the concept AssemblyWithSwitch subsumes the concept AssemblyWithTwoSwitches because in every possible interpretation, an instance of AssemblyWithTwoSwitches must also be an AssemblyWithSwitch.

A realistically scaled conceptual model can include hundreds of related concept definitions. In description logics, concept definitions are automatically organized into a generalization hierarchy; a process sometimes called *classification*. The programmer/knowledge engineer is allowed to specify links in the generalization hierarchy by naming other concepts in a concept definition. Description logics may deduce the additional links in the generalization hierarchy by analyzing the terms of each concept definition and finding implicit relationships. If, for example, the programmer tries to link two concepts with terms that contradict each other, CLASSIC will detect the incoherence and report the problem.

**Table 1.** *Description logic syntax and semantics*

| Syntax | Interpretation |
|---|---|
| TOP | $\Delta^I$ |
| NOTHING | $\emptyset$ |
| (and C D) | $C^I \wedge D^I$ |
| (or C D) | $C^I \vee D^I$ |
| (not C) | $\Delta^I \setminus C^I$ |
| (all p C) | $\{d \in \Delta^I \mid p^I(d) \subseteq C^I\}$ |
| (some p C) | $\{d \in \Delta^I \mid p^I(d) \wedge C^I \neq \emptyset\}$ |
| (at-least n p) | $\{d \in \Delta^I \mid \mid p^I(d) \mid \geq n\}$ |
| (at-most n p) | $\{d \in \Delta^I \mid \mid p^I(d) \mid \leq n\}$ |
| (exactly n p) | $\{d \in \Delta^I \mid \mid p^I(d) \mid = n\}$ |
| (fills p b) | $\{d \in \Delta^I \mid p^I(d) \supseteq b\}$ |
| (one-of $b_1 \ldots b_m$) | $\{b_1, \ldots, b_m\}$ |

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press




```
;; concept definition
(define-primitive SimpleAssembly
   (and
    (all power PowerPacks)
    (at-most 1 power)
    (all switch SwitchPacks)
    (at-most 4 switch)
    (all interface InterfacePacks)
    (at-most 1 interface)
    (all perf-monitor MonitorPacks)
    (at-most 1 perf-monitor)))
;; subconcept with assertion or rule
(define-concept AssemblyWithSwitch
   (and
    SimpleAssembly
    (at-least 1 switch)))
;; individual definitions
(define-individual ASSEMBLY-1
    SimpleAssembly)
;; assertion on individual
(ind-add ASSEMBLY-1 (exactly 1 switch))
```

**Fig. 1.** A simple model for an assembly with slots for switching, power, interface, and performance monitoring circuit packs.

The third expression on the right of Figure 1 defines an individual `ASSEMBLY-1` that is of type `SimpleAssembly`. Individuals inherit from their ancestors just as concepts do, but they are required to be terminal nodes (they cannot have children). The rules of inheritance apply to individuals just as they do to concepts, and individuals are automatically classified just as concepts are. However, certain operations (further explained below), such as rule firing and propagations, are permitted only on individuals.

There are four relations shown in Figure 1. A CLASSIC *role* represents a binary relationship[3] between individuals. For example, `ASSEMBLY-1` would be related to other individuals by the `power`, `switch`, `interface`, and `perf-monitor` roles. For example, we might create a `PERF-MONITOR-1` individual that *fills* the perf-monitor role. It is also possible, and desirable, in CLASSIC to add *value restrictions* that define the kinds of individuals that can be related. In Figure 1, the clause (`all power PowerPacks`) is an example of a value restriction. Value restrictions are also concept descriptions. If, for example, C1 includes the value restriction (all power PowerPacks), then if I1 is an instance of C1 and if I2 fills the power role in I1, then I2 must be a PowerPack.

[3] Some description logics support higher-ordered role relationships as well. All description logics support at least binary relationships.

## 2.2. Dependency maintenance, retraction, and explanation

Description logics can be thought of as active data bases of facts—they are capable of computing the deductive closure of the *told information* (information supplied by a user or some other external source).

The description logic provides the framework for a small set of well-defined logical inferences. The most important inference is *inheritance*. For example, `ASSEMBLY-1` in Figure 1 inherits all the properties of the `SimpleAssembly` concept. A second important inference is *propagation*. Propagation takes place *via* value restrictions on roles, as discussed above. In terms of Figure 1, any individual inserted into the switch role is forced to become a `SwitchPack` because of a propagation along the role `switch` on the individual `ASSEMBLY-1`. If there is something about that individual that conflicts with the definition of SwitchPack, a contradiction results, and an error message is generated.

Facts that are derived from the active part of a CLASSIC model are usually referred to as *derived* information, as contrasted to *told* information, which is directly supplied by a user.

A third important calculation that is part and parcel with description logic systems is contradiction detection. This can be as simple as checking consistency of a concept description. For example, the system would not allow the concept definition

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

```
(define-concept Power1
  (and
    (at-least 1 power)
    (at-most 0 power)))
```

because it is not possible to have *n* fillers for power where *n* is both greater than or equal to 0 and less than or equal to 1 simultaneously. Description logic reasoners only allow input of coherent descriptions in the conceptual model.

Perhaps the most important role of contradiction detection for configuration applications is detecting over-constrained models and rolling back such models to a consistent state. For example, going back to Figure 1, adding the following information to ASSEMBLY-1

```
(ind-add ASSEMBLY-1 (at-least 2 interface) )
```

makes ASSEMBLY-1 inconsistent with its parent concept SimpleAssembly, which has (at-most 1 interface). The description logic reasoner would detect this inconsistency and revert back to the state prior to the addition of (at-least 2 interface) to ASSEMBLY-1. Thus, description logics only allow consistent individuals in the conceptual model.

CLASSIC maintains a set of dependency records for new facts that have been *derived* as a result of the active part of a CLASSIC model. These records are used to support automatic knowledge base reversion to consistent states as well as *retraction* and *explanation*. Users are allowed to retract any piece of information that the system has been told. Anything derived from information included in such a user request is also retracted. Derived information is never allowed to be directly retracted.

Also, any information that has been deduced in CLASSIC can be explained. Explanation uses CLASSIC's dependency records, supplemented by certain additional information computed on the fly, to describe to a user why and how a new fact was derived. This includes explaining how a contradiction was detected. For a more detailed description of explaining these kind of systems see McGuinness and Borgida, 1995; and McGuinness, 1996. As we shall argue subsequently, the effective use of both retraction and explanation are critical to the continued success of the PROSE applications.

### 2.3. Rules

Description logics used in practice allow encoding of forward chaining rules. Rules are associated with concepts, but are applied only to individuals. They are considered to be part of conceptual models by some theorists and not by others (Buchheit et al., 1994a,b). We think of rules as being part of our models because they play such an important role in the PROSE application.

Using CLASSIC, we would add a rule named Power-Rule to the AssemblyWithSwitch concept as follows:

```
(add-rule AssemblyWithSwitch Power-Rule
 (and (at-least 1 power)
      (at-least 1 interface)))
```

An English language version of the Power-Rule is: "any simple assembly with a switch pack must also have a power pack and an interface". This has an antecedent ("any SimpleAssembly with a switch pack") and a consequent ("must have a power pack and an interface"). The antecedent is nothing more than a classified concept, whereas the consequent is an expression in the CLASSIC description language. When an individual is classified under the antecedent, the consequent is asserted on that individual.

Rules are used to model information that should not be used for recognition but is information that is true by virtue of the classification. A prototypical example of this relates to a United States law that people must have social security numbers. Thus, by virtue of being a US Citizen, someone should also have a social security number. However, we would not want an application to demand facts about an individual's status with respect to having a social security number before the application could deduce that the individual is a person.

Also, rules can be used to model circularities. In an effort to maintain faster computation, the language has been restricted so that every concept must be defined before it is used. However, if the knowledge engineer truly has a circular domain, it can be modeled using rules. The rules do not impact the computational power since they are not used for classification of concepts, they are only fired once recognition is performed.

It is possible to perform a data-driven computation using CLASSIC rules. Each time an individual is modified, it is immediately reclassified, and therefore additional parent concepts may be discovered. If any of these new parents have rules, they are applied to that individual. This causes the individual to be reclassified once again, possibly causing additional deductions of parents and associated rule firings. This process repeats itself until there are no new rules to be applied following reclassification. The underlying description logic plays an important role in disciplining the application of rules. It is not possible, for example, for one rule to contradict a previous rule applied to an individual.

Rules can be used to both enforce and check constraints on an object. The power-rule forces any SimpleAssembly with a switch pack to have both power and interface fillers. This is active constraint generation.

In this section, we will discuss the power and value of description logics from a practical standpoint. From a configuration applications standpoint, the most important inferences are inheritance, propagation, user-defined rule firings, and contradiction detection.

Figure 1 provides a simple example of how inheritance and specialization combine to provide new information. The AssemblyWithSwitch concept inherits the number restriction from SimpleAssembly (at-most 4 switch) and

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

combines it with the restriction (`at-least 1 switch`). The resulting number restriction is that number of fillers of the role switch must be between 1 and 4.

Second, description logics also support propagation along roles. For example, in our example (Figure 1) whenever the interface is filled on an instance of `SimpleAssembly`, information is propagated onto the filler. In other words, the interface filler becomes an instance of `InterfacePacks`.

We can also see an example of rule application in the example above. `ASSEMBLY-1` would first be recognized to be a `SimpleAssembly` with a switch. Thus, the Power-Rule would fire, forcing `ASSEMBLY-1` to have at least one filler for both the `power` and the `interface` role.

Then, suppose a second individual were created

```
(define-individual ASSEMBLY-2
  (and
    SimpleAssembly
    (at-most 0 power)))
(ind-add ASSEMBLY-2 (at-least 2 switch))
```

When ASSEMBLY-2 is created, it is in a consistent state. However, the additional information that it has at least two fillers for its switch role, added in the second statement above, allows the description logic to recognize that it is an instance of the antecedent of the Power-Rule. In turn, this causes the power-rule to fire—resulting in an inconsistency because ASSEMBLY-2 was known to have no power fillers, yet the power rule requires at least one power filler. The description logic would detect the inconsistency and revert back to a consistent state prior to the addition of (`at-least 2 switch`).

The examples we have shown are fairly obvious. However, in real applications there are typically hundreds and hundreds of assertions, and detection of contradictions is an extremely valuable feature.

## 2.4. Discussion

When appropriately used, conceptual modelling helps manage the complexity inherent in software applications by providing a sensible, well-organized domain model. In addition, the use of explicit models has the effect of enabling new kinds of functionality not possible with other approaches.

### *2.4.1. Increasing the design space*

All of the product knowledge contained in a PROSE model can be interactively accessed, publicly examined, verified, explained, and manipulated as data. This, in a sense, opens up the design space by making possible new kinds of functionality. We will illustrate this with a single example—the possibility of combining individual configurators into higher-level network *solutions* configurators.

There is a family of fiber optic transmission products available in the market today that can be combined into something called a two-fiber ring network. Such networks are likely to play a key role in the forthcoming information superhighway. A large two-fiber ring network could have more than 50 nodes with constraints that extend throughout the network. Typically, an engineer lays out the network at a high level, then uses a product configurator to configure each node separately. Many of the inputs needed to configure each individual node really reflect characteristics of the network as a whole, or they may be derived from constraints placed on the node by adjacent nodes.

Our modelling techniques can be extended to include the rules of composition for such multinode configurations. Equally important, the models for individual nodes and networks could be combined such that there would be no need for a separate configuration step for individual nodes. Information needed to configure individual nodes would be derived from the higher-level network model, allowing the engineer to concentrate on network design as opposed to making sure the individual node configurations were consistent and valid.

This approach provides a nice, modular way for a knowledge engineer to extend only one portion of the knowledge base, essentially ignoring other portions of the domain, and yet getting the benefits of consistency checking across the entire domain.

### *2.4.2. Dealing with complexity and change*

No matter what implementation technique is selected, configurator applications rely on having some way to encode and represent knowledge about products. For example, when a compiled, procedural language and traditional programming techniques are used, product knowledge is frequently represented as a decision tree using nested case statements. It is hard to appreciate the inappropriateness of such methods without experiencing them first hand. For example, prior to the deployment of our platform, there were many attempts to develop configurators—many of which were deployed successfully but at a certain cost. One of these configurators contained a nested case statement in C that spanned more than 80 pages of source code. It was unmaintainable and quite problematic. One of the authors eliminated this maintenance problem by rewriting the code in CLASSIC. Encoding the case in CLASSIC moved much of the inference burden to the system and away from the human maintained code, thereby decreasing code lines and increasing reliability and maintainability.

Clearly, there is an upper limit on the problem complexity that can be addressed with traditional programming techniques. As a practical matter, developers often sidestep the issue by asking end-users to solve selected parts of a problem—usually the most difficult parts. Although this is a practical solution of sorts, it requires sophisticated product knowledge on the part of the users, and it greatly constrains the population that can effectively use the application.

The requirement for frequent change and updating is also part of the equation. As mentioned earlier, it is not uncommon for 40 to 50% of the product knowledge underlying a

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

configurator to change over the course of a single year. To make things worse, the most successful products change the most and they have the longest effective lifetimes. Hence, the natural tendency of even the best programs to lose their organization over time is accelerated. In most cases, the mapping between the application domain and the underlying software model is complex to start with, and it grows more complex and arbitrary over time.

In contrast, conceptual models are much closer to the way people think about and understand products. In our experience, this substantially increases the range of problems that developers are willing to take on and, just as importantly, it helps them maintain a sensible organization for their application software. For example, there is no need for a developer to maintain complex control structure. Further, inspections of the models are more likely to be meaningful to product experts who do not have programming experience. Today, in fact, nonprogrammers maintain most of the PROSE models.

In a real sense, the modular declarative representation facilitates the development of new product configurators. A typical new configurator will often share much of the high-level structure of other configurators while adding a new subconcept product in the knowledge base. Our experience with knowledge acquisition on the PROSE project shows that constructing a simple question and answer interface to support maintenance by nonprogrammers is quite feasible. Typically, maintenance of a configurator involves adding new subconcepts with appropriate role restrictions. The interfaces for incremental knowledge acquisition tasks may always be somewhat domain-specific, but there is still general work in description logics needed before such tasks can be supported to our satisfaction. For example, when a new subconcept is added, the knowledge should determine what questions users must answer and help them understand the constraints on and consequences of those answers. Gil and colleagues (Gil & Melz, 1996) provide a very nice knowledge acquisition environment for description logic-based planning applications. We have also provided a domain-specific configuration knowledge acquisition tool.

### *2.4.3. Debugging environments*

Most experienced developers are familiar with an environment in which every detail of an application program is under programmer control—from allocating memory to pointer manipulation. The active nature of description logics is a strength from a consistency checking perspective but may be a challenge from a debugging perspective. A description logic will automatically deduce all logical consequences of input and thus it may appear to a knowledge engineer that deductions are being made that were not specified. Helping developers understand events within these models is critical to successfully moving this technology from the world of research into real applications.

We want developers to be able to concentrate on the application domain rather than the underlying technology. We think that explanation and visualization are two promising techniques that will help make the advantages of conceptual modelling available to a wider range of people.

It is both a strength and a weakness of configurators built on description logics that a large number of inferences are performed automatically by the system. Any one of these inferences may be the key to understanding why something did (or did not) happen.

The main challenge to an explanation facility is to explain just the right inference in a language that the user understands. This is complicated by the fact that users are frequently unable to articulate a precise question. Some description logics provide forms of explanation, but to date, CLASSIC's explanation facility is by far the most extensive. CLASSIC provides an underlying logical foundation on which any deduction can be explained, including error deductions.

We have provided a system that provides explanations when the user is able to formulate a precise question, that is capable of suggesting meaningful questions when a user is stuck, and can prune answers to include only the most important parts of a deduction. We also have worked with help desk personnel to determine the most important inferences in configuration applications and then produced an implementation of explanation that explained only those inferences. This experience suggests to us that there may be a tradeoff between completeness and understandability on the part of users. Attempting to explain a complete set of inferences may overwhelm the typical user. On the other hand, explaining too few inferences will not provide enough information for effective debugging. In one of our implementations, we were able to explain more than 80% of the naturally occurring questions posed by knowledge engineers explaining only four major inferences. A flexible, user-tunable system is best of course, but it is important as well to limit output and provide well-chosen defaults governing those limitations.

The existing explanation module (Resnick et al., 1993; McGuinness, 1996) in CLASSIC is aimed at knowledge engineers, but it provides mechanisms for allowing knowledge engineers to specify the presentation form of explanations. In some cases, we have provided natural language templates for the final applications, and we have experimented with graphical representations.

Because they are based on binary relationships between individuals, conceptual models lend themselves to visualization. The graph shown in Figure 1 has a very close correspondence to the underlying constructs of the model, so close that it would be possible to provide a direct manipulation interface in which users inserted switch pack individuals into slots or roles on an assembly individual using drag and drop. A graph showing individuals and their relations conveys a great deal of information about the state of a configuration in a very efficient way.

Of course, there are many challenges to good visualization solutions. Taxonomies might be highly interconnected, there may be many types of objects and connections be-

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

tween objects, and the system may be active and have to be updated. We are currently exploring visualization techniques that can help a user understand what is happening inside an active data base. We are not alone in our efforts at visualization. Some efforts are underway to represent generic frame systems (Karp et al., 1995) and some description logics provide a general graphical interface such as the LOOM interface (Swartout et al., 1996) and Welty's work on CLASSIC (Welty, 1996). Others provide graphical interfaces aimed at particular application domains (Rector et al., 1997). Others have provided graphical presentations of description logic-based data mining applications (Brachman et al., 1993) and we have provided graphical presentations of some configuration applications (McGuinness et al., 1995).

## 3. ANALYSIS OF DESCRIPTION LOGICS FOR CONFIGURATION

We began with a list of needs for our configuration task and now we will summarize how our description logic-based solution meets these needs.

### 3.1. Object-oriented modelling

Description logics have been designed to support representing and reasoning with objects, portions of objects, constraints on portions of objects, relationships between objects, etc. Their frame structured nature along with their support for inferences relating objects to each other makes them a natural choice when this need arises. They can represent objects just as many of the other object-oriented competitors, yet they provide inferences that many competitive systems do not provide. We have shown how a simple object-centered problem depicted with the aide of Figure 1 can be represented in a description logic.

Additionally, the ability to represent more advanced semantic notions such as disjoint classes, hierarchies of roles (such as relative being a more general binary relationship than uncle), inverse roles (such as husband and wife), number restrictions, etc., make description logics particularly well suited to model more complicated object relationships.

### 3.2. Incrementally evolving specifications

One of the design goals for description logics was to be able to add specifications at any point. They deduce all consequences of the information they have at any given time but do not assume that information that is not known yet must be false. Thus, if some particular assembly is only given a restriction on a `power` role but not on a `switch` role, the system will just make deductions based on current knowledge and provide a partial description of the assembly without making negative conclusions about missing switch information. This gives consumers the ability to provide partial specifications and then inspect the logical implications of input. It also allows consumers to generate a partial configuration description and use that as a query against a knowledge base of past completed specifications and retrieve all completed systems matching the partial description. Finally, it allows maintainers of the knowledge base to input information as it becomes available.

### 3.3. Extensible Schemas

New concepts may be added at any time. If a new class is added, the other objects in the knowledge base will be reclassified with respect to the new definition. Thus, when we added `AssemblyWithTwoSwitches`, the system determined that it was subsumed by `AssemblyWithSwitch`. Also, the instances of `AssemblyWithSwitch` such as the individual ASSEMBLY-1 would be reexamined to see if they might be instances of the new concept. This incremental addition capability is particularly important when new products are added to a configurator family. Data-base-oriented solutions do not have this property.

### 3.4. Active completion of knowledge

Since description logics deduce all logical consequences of given information, they are naturally suited to applications where one might put in a small number of restrictions and then ask "what are the consequences of my restrictions?" For example, we may not know much other than that the assembly will require at least one power filler and one interface filler, but if one knows price ranges on those components, then one knows a minimum price of the system so far. More typically, one choice for one filler of a slot in one part of the configuration may have many implications about other choices in the configuration. Thus, a description logic-based system can easily be used to see how any particular choice (or set of choices) impacts the possibilities for other portions of a configuration. Other technologies may do this too, such as constraint-based approaches, but they may not have a declarative specification available for the (possibly incremental) explanation of the deductions.

Another thing that can be useful with inference completion is a deduction of the most specific statement of information that can be logically proven at any moment in time. For example, if we know that all instances of `SwitchPacks` have a price between $X$ and $Y$, and we know a system has exactly $N$ `switches`, all of which are `SwitchPacks`, then we know the system has a switch cost between $N*X$ and $N*Y$. As the system evolves, this price bound can be maintained as it becomes more specific and as other information evolves. Also restrictions on roles become more specific as the session progresses. Initially, we may know that a role needs to be filled with an instance of a `SwitchPack`. This description can be used to query the data base for possible fillers. Later, additional restrictions may have caused this role to be known to be filled with an instance of a `HighCapacitySwitch-Pack`. The more restrictive query can then be submitted to the data base to retrieve a smaller set of possible fillers.

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

### 3.5. Reasoning in the presence of incomplete information

Description logic computation is considered to be eager (as opposed to lazy) and thus consequences are computed for each new piece of information. Thus, incomplete specifications do not present a problem for deduction, since the derived information about the final configuration specification incrementally becomes more detailed.

### 3.6. Detect and maintain consistency

Since description logics provide a sort of theorem proving capability by providing logical completion of information, they detect inconsistencies as soon as they are logically implied by the given information. This can be particularly useful if one type of customer for the configuration system wants to input a partial or complete parts list for a price quote and validation. The description logic-based system can detect if the parts are consistent when put together into a configuration. If there is an inconsistency, the system can identify the conflicting information. CLASSIC, and some other description logics, can provide access to information about the temporary state of the knowledge base when it detected the inconsistency and before it reverted back to a consistent state. Access to this temporary state is required in order to debug conflicting inference chains. Without access to it, it is impossible to determine the exact conflicting deductions.

### 3.7. Retraction and truth maintenance

Since description logics maintain a consistent knowledge base at all times and since they keep records of what information was deduced as a result of other information, they can easily support applications where information is added, consequences are deduced, information is later retracted, and then consequences are simultaneously retracted. They also will not allow the situation that may cause problems in expert systems applications where one rule fires producing assertion X and a later rule fires, producing assertion ∼X.

### 3.8. Declarative encoding of knowledge

Instead of requiring domain-specific inferences to be recorded procedurally, for example in procedures, description logics provide a foundation for representing knowledge declaratively. As a result, they can provide a more modular and more readable knowledge base of information for knowledge engineers to maintain. This facilitated the large savings in the previously mentioned 80-page case statement by eliminating the code dealing with procedural information. Also, this declarative specification can then be used to support explanations of deductions. Explanation of reasoning was not one of the initial project goals; however, it emerged as the project grew. To support knowledge maintenance, it became critical to provide automatic explanation facilities. Also, the customer help desk required justifications of conclusions.

Although it was not one of our initial needs, description logics can also be used to identify when a configuration is complete. Weida provides a description of how his description logic-based configuration design identifies when all parts have been specified completely enough to be sent in as an order (Weida, 1996).

Beyond these features, it has been stated that one of the greatest advantages of using a description logic is the support for conceptual modelling. These claims are not restricted to configuration domains. Such claims have been made in process engineering (Baader & Sattler, 1996), medical applications (Rector et al., 1997), natural language (Bagnasco et al., 1996; Kuessner, 1997), functional modelling (Compatengenlo et al., 1997), as well as configuration. The first round of proof is the growing empirical evidence that families of applications can be built, maintained, and supported in multiple domains. In the next round of proof, we expect to see groups reuse knowledge by adding new layers to preexisting knowledge bases, perhaps originally built by other individuals and/or groups.

### 3.9. Limitations of description logics (with respect to configuration)

There are some things for which description logics, in their current implementations, have not been designed. We would like to address the topics and then discuss the impact on configuration problems.

#### *3.9.1. Probability representation*

While proposals have been published on probabilistic extensions to description logics (e.g., Jaeger, 1994; Koller et al., 1997), implemented description logics do not include a representation and reasoning formalism for probabilities. This did not impact our representation of the configuration task.

#### *3.9.2. Preference semantics*

Having a method of representing preferences would have made life easier in many cases. For example, we encountered statements such as "Boards of type X go into slot 0 first, slot 11 next, and slot 21 last" with an increasing frequency over the years. This is especially true for equipment that must go through a traffic engineering phase. Traffic engineering is used to balance load across resources.

#### *3.9.3. Completion support*

While description logics perform logical completion (i.e., they make all implicit information explicit), they do not support a formal notion of determining all roles on a particular object whose at-least restrictions are greater than the number of fillers for the role and then generating fillers for those roles. It is one of the strengths of description logics for incremental tasks that they reason with the open-world

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

assumption. However that leads to descriptions of configurations where users have not specified enough information for the configuration system to deduce a complete model of the component to be configured. Additionally, in most description logics, domain-independent choice of an order of roles to fill on objects in a class is not a simple task. Current configuration implementations force the knowledge engineer to determine an order of role filling to be used. Thus, the knowledge engineer in the stereo domain might decide that he or she should fill the receiver role first since it places more constraints on fillers for other roles, and then fill the speaker role, followed by the cd-player role. In a deployed configurator, it took one of our knowledge engineers a week to determine the proper ordering for role closure. A more supportive deductive system might attempt to determine a role order that could be used to close roles. (We are currently addressing this issue with theoretical work aimed at determining when knowledge bases may produce an ordering of roles that can be used to obtain the same minimal model of the component to be configured.) In our implemented complete functions, we used domain knowledge to choose an order for role closure and then wrote functions that would close roles on any object in given classes.

#### 3.9.4. *Optimization*

Implemented description logics do not support an optimization component that may be used to choose component choices that are not provably implied by the specification. This did not cause any problems for our configurators since the system implemented a sort of default choice which was used to complete roles whenever the user did not specify enough information initially. While an optimization component for configuration may be desirable, we conjecture that implemented configurators may produce better solutions if they interface to special-purpose optimization routines that are knowledgeable of the domain.

#### 3.9.5. *Limited expressive power*

Description logics have consciously chosen to limit the constructors in their languages in order to be able to support complete (or near complete) reasoning in an acceptable time. Thus, it could be the case that particularly complicated products might push the representational component to its limits. In our eight years of modelling experience, we were not adversely impacted by the representational restrictions of choosing one of the less expressive description logics. In CLASSIC, there is a test function that allows a knowledge engineer to define additional functions that will encode information that goes beyond the base language. It was by analyzing the use of these test functions that we determined how the core description logic should expand over the years. What we found through this analysis and through interviews with users, was that people rarely thought additional expressive power was necessary to complete their tasks. The most common requests of the basic system usually had to do with additional environmental support (such as graphical browsers, knowledge acquisition environments, etc.) and rarely had to do with additional constructors.

## 4. CONCLUSION

Although it is not often recognized as such, modelling activities of some sort underlie much of what is today called application programming. However, the mapping between an application domain and a software implementation is usually complex, and it loses its structure over time as a system is updated. In fact, every successful project has a few gurus around who are indispensable largely because they understand the complex relationship between the application domain and the application software. The description logic-based modelling techniques discussed above are an attempt to bring the structure and vocabulary of the underlying software model closer to the way people think about an application domain. The techniques represent a convergence of work in several disciplines, namely data modelling, programming languages (primarily object-oriented technology), and artificial intelligence (knowledge representation and knowledge engineering). Perhaps because the physical structure of hardware products provides such a compelling model, configurators have provided fruitful ground to exercise these techniques in a real application.

## ACKNOWLEDGMENTS

We have greatly benefited from collaboration with Charlie Foster, Gregg Vesonder, Steve Solomon, Harry Moore, Ron Brachman, Peter Patel-Schneider, Lori Alperin Resnick, as well as many members of their teams.

## REFERENCES

Ackroff, J.M., Surko, P., Vesonder, G.T., & Wright, J.R. (1990). SARTS AutoTest-2. In *Practical Experience in Building Expert Systems*, pp. 209–226. John Wiley & Sons, New York.

Baader, F., & Sattler, U. (1996). Knowledge representation in process engineering. *Proc. Int. Workshop on Description Logics*, pp. 74–79. Cambridge (Boston), Massachusetts.

Bagnasco, C., Bresciani, P., Magnini, B., & Strapparava, C. (1996). *Natural language interpretation for public administration database querying in the TAMIC demonstrator*. Technical Report 9601-09, IRST. Povo, TN, Italy, January 1996. Also published in *Proc. of the NLDB'96 Workshop*, Amsterdam.

Booch, G. (1996). The unified modelling language. *Unix Review (41)*, 41–48.

Borgida, A., Brachman, R.J., McGuinness, D.L., & Resnick, L.A. (1989). CLASSIC: A structural data model for objects. *Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, June, 1989, pp. 59–67.

Brachman, R.J. (1979). On the epistemological status of semantic networks. In *Associative Networks: Representation and Use of Knowledge by Computers*, (Findler, N.V., Ed.), pp. 3–50. Academic Press, New York.

Brachman, R.J. (1983). What ISA is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer 16(10)*, 30–36.

Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A. & Borgida, A. (1991). Living with CLASSIC: when and how to use a KL-ONE-like language. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, (Sowa, J.F., Ed.), pp. 401–456. Morgan Kaufmann, San Mateo, California.

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press

Brachman, R.J., & Schmolze, J.G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science 9(2)*, 171–216.

Brachman, R.J., Selfridge, P., Terveen, L., Altman, B., Borgida, A., Halper, F., Kirk, T., Lazar, A., McGuinness, D.L., & Resnick, L.A. (1993). Integrated support for data archaeology. *International Journal of Intelligent and Cooperative Information System 2(2)*, 159–185.

Brodie, M.L., Mylopoulos, J., & Schmidt, J.C. (Eds.) (1982). *Conceptual Modelling*. Springer-Verlag, New York.

Brooks, F., Jr. (1975*). The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts.

Buchheit, M., Donini, F.M., Nutt, W., & Schaerf, A. (1994*a*). Refining the structure of terminological systems: Terminology = Schema + Views. *Proc. Am. Assoc. of Artificial Intelligence*, pp. 199–204.

Buchheit, M., Jeusfeld, M., Nutt, W., & Staudt, W. (1994*b*). Subsumption between queries to object-oriented databases. *Information Systems 19(1)*, 33–54.

Compatengenlo, E., Donini, F., & Rumolo, G. (1997). A description logic for reasoning with behavioural knowledge. *Proc. Int. Workshop on Description Logics*, pp. 44–48. Gif sur Yvette (Paris), France.

Giarrantano, J., & Riley, G. (1994). *Expert Systems: Principles of Knowledge-based Systems*. PWS-Kent Publishings, Boston, Massachusetts.

Gil, Y., & Melz, E. (1996). Explicit representations of problem-solving strategies to support knowledge acquisition. *Proc. Thirteenth National Conf. on Artificial Intelligence*, Portland, Oregon, pp. 469–476.

Jaeger, M. (1994). Probabilistic reasoning in terminological logics. *Proc. Fourth Int. Conf. on Principles of Knowledge Representation and Reasoning*, Bonn, Germany, pp. 305–316.

Karp, P.D., Myers, K., & Gruber, T. (1995). The generic frame protocol. *Proc. 1995 Int. Joint Conf. on Artificial Intelligence*, pp. 768–774.

Koller, D., Levy, A., & Pfeffer, A. (1997). P-CLASSIC: A tractable probabilistic description logic. *Proc. Fourteenth Nat. Conf. on Artificial Intelligence*, pp. 390–397.

Kuessner, U. (1997). Applying DL in Automatic Dialogue Interpreting. *Proc. Int. Workshop on Description Logics*, pp. 54–58. Gif sur Yvette (Paris), France.

Loucopoulos, P., & Zicari, R. (Eds.) (1992). *Conceptual Modelling, Databases, and CASE*. John Wiley & Sons, New York.

MacGregor, R.M. (1991). The evolving technology of classification-based knowledge representation systems. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, (Sowa, J., Ed.), pp. 385–400. Morgan Kaufmann, Los Altos, California.

McDermott, J. (1981). R1: The Formative Years. *AI Magazine*, 2*(2)*, 21–29.

McGuinness, D.L. (1996). *Explaining reasoning in description logics*. Ph.D. Thesis, Department of Computer Science, Rutgers University. Also available as a Rutgers Technical Report No. LCSR-TR-277.

McGuinness, D.L., & Borgida, A. (1995). Explaining subsumption in description logics. *Proc. Int. Conf. on Artificial Intelligence*, pp. 816–821. Montreal, Canada.

McGuinness, D.L., Resnick, L.A., & Isbell, C. (1995). Description logic in practice: A CLASSIC application. *Proc. Int. Conf. on Artificial Intelligence*, pp. 2045–2046. Montreal, Canada.

McGuinness, D.L., & Wright, J.R. (1998). An industrial strength description logic-based configurator platform. In *IEEE Expert, Special Issue on Configuration*, Faltings, B. and Freuder, G., (Eds.).

Owsnicki-Klewe, B. (1988). Configuration as a consistency maintenance task. *Proceedings of GWAI-88—the 12th German Workshop on Artificial Intelligence*, pp. 77–87. Springer Verlag, New York.

Patil, R., Fikes, R., Patel-Schneider, P., Mckay, D., Finin, T., Gruber, T., & Neches, R. (1992). The DARPA knowledge sharing effort: Progress report. *Proc. Third Int. Conf. on Principles of Knowledge Representation and Reasoning*, pp. 777–788. Cambridge, Massachusetts.

Rector, A.L., Bechhofer, S.K., Goble, C.A., Horrocks, I., Nowlan, W.A., & Solomon, W.D. (1997). The GRAIL concept modelling language for medical terminology. *Artificial Intelligence in Medicine 9*, 139–171.

Resnick, L.A., Borgida, A., Brachman, R.J., McGuinness, D.L., & Patel-Schneider, P.F. (1993). *CLASSIC Description and Reference Manual for the COMMON LISP Implementation: Version 2.1*. AI Principles Research Department, AT&T Bell Laboratories, New Jersey.

Rychtyckyj, N. (1996). DLMS: An evaluation of KL-ONE in the automobile industry. *Proc. Fifth Int. Conf. on Principles of Knowledge Representation and Reasoning*, Cambridge, Massachusetts, pp. 588–596.

Sowa, J. (1991). *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, Morgan-Kaufmann, New York.

Swartout, B., Patil, R., Knight, K., & Russ, T. (1996). Toward distributed use of large-scale ontologies. *Proc. Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, pp. 403–409. Banff, Alberta, Canada.

Vesonder, G.T., Stolfo, S.J., Zielinski, J.E., Miller, F.M., & Copp, D.H. (1983). ACE: An expert system for telephone cable maintenance, *Int. Joint Conf. on AI*, *8*, 116–120.

Weida, R. (1996). Closed terminologies in description logics. In *Proceedings of the AAAI Fall Symposium Series on Configuration*, (Faltings, Freuder, Haselboeck, MacCallum, McGuinness, and Mittal, Eds.) pp. 11–18. Cambridge Massachusetts.

Weixelbaum, E. (1991). *The C-Classic Reference Manual Release 1.0*. AT&T Bell Laboratories, New Jersey.

Welty, C. (1996). An HTML Interface for Classic. *Proc. 1996 Int. Workshop on Description Logics*, 200–202. AAAI Press.

Woods, W.A. (1975). What's in a link: Foundations for semantic networks. In *Representation and Understanding: Studies in Cognitive Science*, (Bobrow, D.G. and Collins, A.M., Eds.), pp. 35–82. Academic Press, New York.

Wright, J.R., Zielinski, J.E., & Horton, E.A. (1988). Expert systems development: The ACE System. In *Expert System Applications in Telecommunications*, (Jay Liebowitz, Ed.), pp. 45–72. John Wiley & Sons, New York.

Wright, J.R., Weixelbaum, E., Vesonder, G.T., Brown, K.E., Palmer, S.R., Berman, J.I., & Moore, H.H. (1993). A knowledge based configurator that supports sales, engineering, and manufacturing at AT&T Network Systems. *AI Magazine 14(3)*, 69–80.

---

**Deborah L. McGuinness** has just accepted the position of associate director and senior research scientist of the knowledge systems laboratory of Stanford University. For the previous 18 years, she has been a researcher for AT&T Labs and Bell Labs. Her research interests include usability issues in deductive systems (particularly inference explanation, meta languages for pruning and presentation, and interfaces), description logics, knowledge-enhanced search, knowledge representation and reasoning systems and their applications, conceptual modelling and ontologies, and configuration. Her work has been mostly in research but it included rotations into Home Information Systems development and, most recently, managing the emerging technologies and opportunities group in the Personal Online Services Division. She received a Ph.D. in Computer Science from Rutgers, a M.S. in Computer Science from the University of California at Berkeley, and a B.S. in Computer Science and Mathematics from Duke University. She is a member of AAAI, NYNMA, and ACM.

**Jon R. Wright** is currently a principal member of technical staff in the Online Platforms Research Department at AT&T Labs. Current work is focused on development of new internet services. He joined Bell Telephone Laboratories in 1978 as a Human Factors specialist, and has about 15 years experience as a knowledge engineer in telecommunications, including applications in switching, transmission systems, local loop, and special services. Jon has an undergraduate degree in psychology from the University of Tulsa and a Ph.D. in applied experimental psychology from Rice University. E-mail: jrw@research.att.com

https://doi.org/10.1017/S089006049812406X Published online by Cambridge University Press