"Reducing" CLASSIC to Practice: Knowledge Representation Theory Meets Reality*

Ronald J. Brachman[1], Alex Borgida[2,**],
Deborah L. McGuinness[3], and Peter F. Patel-Schneider[4]

[1] Yahoo! Research
[2] Rutgers University
[3] Rensselaer Polytechnic Institute
[4] Bell Labs Research

Abstract. Most recent key developments in research on knowledge representation (KR) have been of the more theoretical sort, involving worst-case complexity results, solutions to technical challenge problems, etc. While some of this work has influenced practice in Artificial Intelligence, it is rarely—if ever—made clear what is compromised when the transition is made from relatively abstract theory to the real world. CLASSIC is a description logic with an ancestry of extensive theoretical work (tracing back over twenty years to KL-ONE), and several novel contributions to KR theory. Basic research on CLASSIC paved the way for an implementation that has been used significantly in practice, including by users not versed in KR theory. In moving from a pure logic to a practical tool, many compromises and changes of perspective were necessary. We report on this transition and articulate some of the profound influences practice can have on relatively idealistic theoretical work. We have found that CLASSIC has been quite useful in practice, yet still strongly retains most of its original spirit, but much of our thinking and many details had to change along the way.

FORWARD - February 2009

The practical instantiation of theories is an important, but often neglected, component of knowledge representation. The work on CLASSIC, described in this paper, was undertaken at AT&T Bell Labs around 1990, and fits into this component, taking the formal theories of description logic and crafting a useful system for representing information.

The state of the art in description logics has changed considerably since CLASSIC was first envisioned in the late 1980s. Advances in algorithms and computers have made for effective reasoners for theoretically intractable description logics,

* A slightly different version published in *Artificial Intelligence*, 114, October 1999, pages 203–237.

** Supported in part by the AT&T Foundation and NSF IRI 9119310 & 9619979.

A.T. Borgida et al. (Eds.): Mylopoulos Festschrift, LNCS 5600, pp. 436–465, 2009.

leading up to modern reasoners for description logics like SHOIQ, where reasoning is NEXPTIME complete.

These advances in reasoning have led to the adaptation of description logics to the Semantic Web. The changes needed were theoretically quite small, amounting mostly to using IRIs as names and adding the datatypes of XML Schema. This lead, through a number of intermediate steps, to the W3C Web Ontology Language OWL. However, quite a bit of practical adaptation was needed as well, including allowing non-logical decorations, adding a method of combining documents on the web into one knowledge base, building user interface systems like Protégé (http://www.co-ode.org/downloads/protege-x/), and providing APIs for popular programming languages. There are a now multiple reasoning systems for OWL, including Pellet (http://clarkparsia.com/pellet/), Racer (http://www.racer-systems.com/), and FaCT++ (http://code.google.com/p/factplusplus/), which are based on this combination of theoretical underpinnings and attention to practical details, now prevalent in the description logic community.

1 Introduction

In recent years, the research area of knowledge representation (KR) has come to emphasize and encourage what are generally considered more "theoretical" results, such as novel logics, formal complexity analyses, and solutions to highly technical challenge problems. In many respects, this is reasonable, since such results tend to be clean, presentable in small packages (i.e., conference-length papers), easily built upon by others, and directly evaluable because of their formal presentation. Even the recent reappearance of papers on algorithms and empirical analyses of reasoning systems has emphasized the formal aspects of such work. But lurking in the virtual hegemony of theory and formalism in published work is the tacit belief that there is nothing interesting enough in the reduction of KR theory to practice to warrant publication or discussion. In other words, it seems generally assumed that theoretical contributions can be reduced to practice in a straightforward way, and that once the initial theoretical work is wrapped up, all of the novel and important work is done.

Experience in building serious KR systems challenges these assumptions. There is a long and difficult road to travel from the pristine clarity of a new logic to a system that really works and is usable by those other than its inventors. Events along this road can fundamentally alter the shape of a knowledge representation system, and can inspire substantial amounts of new theoretical work.

Our goal here is to attempt to show some of the key influences that KR practice can exert on KR theory. In doing so, we hope to reveal some important contributions to KR research to be made by those most concerned with the reduction to practice. In order to do this, we will lean on our experience with CLASSIC, a description logic-based KR system. CLASSIC has seen both ends of the theory-to-practice spectrum in quite concrete ways: on the one hand, it was developed after many years of KL-ONE-inspired research on description systems, and was initially presented as a clean and simple language and system [5]

with a formal semantics for subsumption, etc.; on the other, it has also been re-engineered from an initial LISP implementation to C and then to C++, has been widely distributed, and has been used in several fielded industrial products in several companies on an everyday basis (making CLASSIC one of a few KR systems to be moved all the way into successful commercial practice). Our substantial system development effort made it clear to us that moving a KR idea into real practice is not just a small matter of programming, and that significant research is still necessary even after the basic theory is in place.

Our plan in this paper is first to give an introduction to the goals and design of our system, and then to summarize the "theoretical" CLASSIC as originally proposed and published. We will then explain our transition from research design to production use. Rather than give a potentially unrevealing historical account, we will subsequently attempt to abstract out five general factors that caused important changes in our thinking or in the system design, giving concrete examples of each in the evolution of CLASSIC:

1. general considerations in creating and supporting a running system;
2. implementation issues, ranging from efficiency tradeoffs to the sheer complexity of building a feature;
3. general issues of real use by real people, such as learnability of the language and error-handling;
4. needs of particular applications; and
5. the unearthing of incompleteness and mistakes through programming, use, and community discussion.

While some of these concerns may seem rather prosaic, the point is that they have a hitherto unacknowledged—and substantial—influence on the ultimate shape, and certainly on the ultimate true value of any knowledge representation proposal. There is nothing particularly atypical about the research history of CLASSIC; as a result, the lessons to be learned here should apply quite broadly in KR research, if not in AI research in general.

2 The Original CLASSIC

While potential applications were generally kept in mind, CLASSIC was originally designed in a typical research fashion—on paper, with attention paid to formal, logical properties, and without much regard to implementation or application details. This section outlines briefly the original design, before we attempted to build a practical tool based on it. The interested reader can also see [5], which is a traditional research article on the original version of the language, and [12] for more details.

2.1 Goals

CLASSIC was designed to clarify and overcome some limitations in a series of knowledge representation languages that stemmed originally from work on KL-ONE [9,13]. Previous work on NIKL [32], KRYPTON [10], and KANDOR [36] had

shown that languages emphasizing structured descriptions and their relationships were of both theoretical and practical interest. Work on KL-ONE and its successors[1] grew to be quite popular in the US and Europe in the 1980's, largely because of the semantic cleanliness of these languages, the appeal of object-centered ("frame") representations, and their provision for some key forms of inference not available in other formalisms (e.g., *classification*—see below). Numerous publications addressed formal and theoretical issues in "KL-ONE-like" languages, including formal semantics and computational complexity of variant languages (see, or example, [19]). However, the key prior implemented systems all had some fundamental flaws,[2] and the CLASSIC effort, initiated in the mid '80's at AT&T, was in large part launched to design a formalism that was free of these defects.

Another central goal of CLASSIC was to produce a compact logic and ultimately, a small, manageable, and efficient system. Small systems have important advantages in a practical setting, such as portability, maintainability, and comprehensibility. Our intention was eventually to put KR technology in the hands of regular technical (non-AI) employees within AT&T, to allow them to build their own domain models and maintain them. Success on this account seemed to very strongly depend on how simply and effortlessly new technology could integrate into an existing environment and on how easy it would be to learn how to use it, not to mention on reasonable and predictable performance. Our plan was to take computational predictability (both of the inferences performed and the resources used) seriously. All in all, because of our desire to connect with extremely busy developers working on real problems, simplicity and performance were paramount in our thinking from the very beginning.

As has been often discussed in the literature [11,21], expressiveness in a KR language trades off against computational complexity, and our original hope was to produce a complete inference system that was worst-case tractable. This contrasts the CLASSIC work even with other contemporaneous efforts in the same family, e.g., LOOM [24] and BACK [39], and with much work elsewhere in KR. Like CLASSIC, both of these systems ended up in the hands of users at sites other than where they were developed (in fact, LOOM has a user base of several hundreds, mostly in the AI research community), and had to go through the additional efforts, described in this paper, at supporting a user interface, escape mechanisms from expressive limitations, and rules. However, both LOOM and BACK opted for more expressive logics and incomplete implementations, which left them with the problem of characterizing the inferences (not) being performed. In CLASSIC, although we eventually incorporated language features that were on paper intractable (e.g., role hierarchies), the implementation of concept reasoning was kept complete, except for reasoning with individuals in concepts, and in that case both a precise formal and procedural characterization were given [7].

[1] These systems came to be called "description logics"—see below.

[2] E.g., NIKL had no mechanism for asserting specific, concrete facts; KRYPTON was so general as to be fairly unusable; and KANDOR handled individual objects in a domain in a very incomplete way.

Many more recent description logic-based KR systems have explicitly taken a different view of this expressiveness/tractability tradeoff. They chose to implement complete reasoners for expressive languages and thus, of necessity, have intractable inference algorithms. Initially these systems had poor computational properties [2,15], but recent advances in description logic inference algoritms, initiated by Ian Horrocks, have led to a new generation of systems with surprisingly effective performance [22,38].

CLASSIC continues to be an important data point, and a useful experiment on exactly how one can design a small but useful language, but as we see below, the original theoretical goal of worst-case polynomial complexity could not be preserved without detriment to real users.[3]

Finally, CLASSIC was designed to fill a small number of specific application needs. We had had experience with a form of deductive information retrieval, for example in the context of information about a large software system [18], and needed a better tool to support this work. We also had envisioned CLASSIC as a deductive, object-oriented database system (see [5]; some success on this front was eventually reported in [42] and [14]). It was *not* our intention to provide some generic "general knowledge representation system," applicable to any and all problems. CLASSIC would probably not be useful, for example, for arbitrary semantic representation in a natural language system, nor was it intended as a "shell" for building an entire expert system. But the potential gains from keeping the system small and simple justified the inability to meet arbitrary (and too often, ill-defined) AI needs—and it was most important to have it work well on our target applications.

2.2 The Description Language

CLASSIC is based on a *description logic* (DL), a formal logic whose principal objects are structured terms used to describe individual objects in a domain. Descriptions are constructed by composing *description-forming constructors* from a small repertoire and specifying arguments to these constructors. For example, using the constructor **ALL**, which is a way to restrict the values of a property to members of a single class, we might construct the description, "(something) all of whose children are female":

```
(ALL child FEMALE).
```

Descriptions can then be asserted to hold of individuals in the domain, associated with names in a knowledge base, or used in simple rules. Because of the formal, compositional structure of descriptions (i.e., they are like the complex types in programming languages), certain inferences follow both generically—one

[3] However, we state emphatically that this did not mean that the only alternative was to resort to an arbitrarily expressive language to satisfy our users. As we discuss below, the extensions made to meet real needs were generally simple, and CLASSIC is still clearly a small system, yet one of the most widely used description logic-based systems.

description can be proven to imply another—and with respect to individuals—a description can imply certain non-obvious properties of an individual.

In CLASSIC, as in most description logic work, we call our descriptions *concepts*, and individual objects in the domain are modeled by *individuals*. To build concepts, we generally use descriptions of properties and parts, which we call *roles* (e.g., `child`, above). CLASSIC also allows the association of simple *rules* with named concepts; these were considered to be like forward-chaining procedural rules (also known as "triggers" in active databases). For example,

```
AT&T-EMPLOYEE ⇝
(AND (AT-LEAST 1 HRID-number)
     (ALL HRID-number 7-DIGIT-INTEGER))
```

would mean "If an AT&T employee is recognized then assert about it that it has at least one HR ID number, all of which are 7-digit integers."

CLASSIC's description-forming constructors were based on the key constructs seen in frame representations over the years. These constructors have cleaned up ambiguities in prior frame systems, and are embedded in a fully compositional, uniform description language. The constructors in the original design ranged from conjunction of descriptions (**AND**); to role "value restrictions" (**ALL**); number restrictions on roles (**AT-LEAST**, **AT-MOST**); a set-forming constructor (**ONE-OF**); and constructors for forming "primitive" concepts (**PRIMITIVE**, **DISJOINT-PRIMITIVE**), which have necessary but not sufficient conditions. CLASSIC also has a constructor **SAME-AS**, specifying objects for which the values of two sequences of roles have the same value; and a role-filling constructor (**FILLS**) and a constructor for "closing" roles (**CLOSE**),[4] although these were originally only applicable to individuals.

2.3 Operations on CLASSIC Knowledge Bases

The description-forming constructors are used to create descriptions that are, in turn, used to define named concepts, create rules, and describe individual objects in the domain. In order to create new concepts and assign descriptions to individuals, etc., the user interacts with the CLASSIC system by means of a set of *knowledge base-manipulating operations*. Generally speaking, operations on a CLASSIC knowledge base (KB) include additions to the KB and queries.

Additive (monotonic) updates to the KB include the definition of new concepts or roles, specification of rules, and assertion of properties to hold of particular individuals. By "definition" here, we mean the association of a name (e.g., `SATISFIED-GRANDPARENT`) with a CLASSIC description (e.g., `(AND PERSON (ALL grandchild MARRIED))`, intended to denote "a person all of whose grandchildren are married"). One of the contributions of CLASSIC, also developed

[4] Role fillers for CLASSIC's individuals are treated under a "non-closed-world" assumption, in that unless the KB is told that it knows all the fillers of a certain role, it assumes that more can be added (unless the number restrictions forbid it, in which case role closure is implied).

contemporaneously in BACK [43], is the ability to specify incomplete information on individuals; for example, it is possible to assert that `Leland` is an instance of

```
(AND (AT-LEAST 1 child)
     (ALL works-for
          (ONE-OF Great-Northern-Hotel Double-R-Diner)))
```

indicating that "Leland has at least one child and works for either the Great Northern Hotel or the Double-R Diner." Thus, individual objects are not required to be in the restricted form of simple tuples or complete records.

CLASSIC can also answer numerous questions from a KB, including whether one concept subsumes another, whether an individual is an instance of a concept, and whether two concepts are disjoint, and it can respond to various types of retrieval queries (e.g., fetch the properties of an individual, fetch all the instances of a concept).

These general operations on CLASSIC knowledge bases were characteristic of the system throughout its evolution, although specific operations changed in interesting ways and new ones (especially dealing with retraction of information) were added, as discussed later.

2.4 Inferences

A key advantage of CLASSIC over many other systems was the variety and types of inferences it provided. Some of the standard frame inferences, like (strict) inheritance, are part of CLASSIC's definition, but so are several others that make description-logic-based systems unique among object-centered KR systems. Among CLASSIC's inferences are

- *"completion" inferences*: logical consequences of assertions about individuals and descriptions of concepts are computed; there are a number of these inferences CLASSIC can make, including *inheritance* (if A is an instance of B and B is a subclass of C, then A "inherits" all the properties of C), *combination* of restrictions on concepts and individuals, and *propagation* of consequences from one individual to another (if A fills a role r on B, and B is an instance of something which is known to restrict all of its fillers for the r role to be instances of D, then A is an instance of D);
- *contradiction detection*: inherited and propagated information is used to detect contradictions in descriptions of individuals, as well as incoherent concepts.
- *classification and subsumption inferences*: *concept classification*, in which all concepts more general than a concept and all concepts more specific than a concept are found; *individual classification*, in which all concepts that an individual satisfies are determined; and *subsumption*, i.e., whether or not one concept is more general than another.
- *rule application*: when an individual is determined to satisfy the antecedent of a rule, it is asserted to satisfy the consequent as well.

2.5 Other Theoretical Aspects of CLASSIC

Since the original design of CLASSIC proceeded mainly from a theoretical standpoint, other formal aspects of the logic were explored and developed. CLASSIC's concept constructors had a traditional formal semantics similar to the semantics of related languages. Since reasoning about individuals was more procedural, and did not have the same formal tradition, it did not initially receive the same precise formal treatment. In hindsight, it would have been of considerable help to have found a formal semantics for this aspect too, although this ended up requiring entirely new machinery (as discovered by Donini, *et al.* [20]) and may well have cost us too much in terms of time and lost opportunities (another key factor in the production of a successful system).

Given our desire for compactness and performance, we were also concerned with the computational complexity of the inferences to be provided by CLASSIC. Based on the experience with implementing previous description logic-based systems, and the work of Aït-Kaci [1] on reasoning with attribute-chain identities, we believed that we had a good chance to develop an efficient, polynomial-time algorithm for the subsumption inference. However, at this stage we did not have a formal proof of the tractability of the logic, nor of the completeness of our implementation. To some extent we were going on faith that the language was simple enough, and that we were avoiding the known intractability traps, such as those pointed out by Nebel [34]. As we shall see, this faith was in part misplaced, from the strictly formal point of view, although in practice we were on the right track.

2.6 Anticipating Implementation

The original CLASSIC proposal to some extent anticipated building a practical tool. Two concessions were made directly in the language. First, inspired by the TAXIS database programming language [33], we allowed the formation of concepts that could compute their membership using *test functions* to be written in the host programming language. **TEST** concepts would act as primitive concepts, in that their necessary and sufficient conditions would not be visible to CLASSIC for inference. But they would allow the user to partly make up for CLASSIC's limited expressive power, at least in dealing with individuals.

Second, we specified a class of individuals in the language called *"host" individuals*, which would allow direct incorporation of things like strings and numbers from the host programming language. Many previous KR languages had failed to make a clean distinction between values, such as numbers and strings, borrowed directly from the implementation, and objects totally within the representation system. CLASSIC cleared this up even in the initial formal design; for example, a host individual could not have roles, since it is immutable and all of its properties are implied by its identity. Also, the (test) definitions of important host concepts (like `NUMBER`, `STRING`, etc.) could be derived automatically from the host environment.

2.7 The Result

Prior to the completion and release of an implementation and its use in applications, CLASSIC thus had the description language grammar illustrated in Figure 1—this is roughly the version of CLASSIC published in 1989 [5], and was the initial basis for discussions with our development colleagues (see below). Using descriptions formed from this grammar, a user could create a knowledge base by defining new named concepts and roles, asserting that certain restricted types of descriptions applied to individuals, and adding rules, as defined in Figure 2.

Numerous straightforward types of KB queries were also available. These included retrieval of information told to the system, retrieval of information derived by the system, and queries about the derived taxonomy of concepts and individuals.

Even at this point, the CLASSIC logic made a number of new contributions to KR, including the following:

- full integration of host-language values,
- support for equalities on attribute chains (**SAME-AS**),
- the ability to assert partial information about individuals,
- the addition of a simple form of forward-chaining implication to a KL-ONE-like language,
- the **ONE-OF** construct for constructing explicit sets of individuals,
- a broad cadre of inferences including full propagation of information implied by assertions, and
- a comprehensive and clean retraction mechanism for assertions,

```
<concept-expression> ::=
      THING | CLASSIC-THING | HOST-THING |                    % built-in names
      <concept-name> |                                  % names defined in the KB
      (AND <concept-expression>+) |                          % conjunction
      (ALL <role-name><concept-expression>) |          % universal value restriction
      (AT-LEAST <positive-integer><role-name>) |         % minimum cardinality
      (AT-MOST <non-negative-integer><role-name>) |     % maximum cardinality
      (SAME-AS (<role-name>+)(<role-name>+)) |          % role-filler equality
      (TEST <function> [<realm>]) |                        % procedural test
      (ONE-OF <individual-name>+) |                      % set of individuals
      (PRIMITIVE <concept-expression> <index>) |         % primitive concept
      (DISJOINT-PRIMITIVE <concept-expression> <group-index> <index>)
<realm> ::= HOST | CLASSIC
<individual-expression> ::=
      <concept-expression> |
      (FILLS <role-name> <individual-name>) |                % role-filling
      (CLOSE <role-name>) |                                  % role closure
      (AND <individual-expression>+)                         % conjunction
```

Fig. 1. The Original CLASSIC Expression Grammar (comments in italics)

```
<knowledge-base> ::= <statement>+
<statement> ::=
        (DEFINE-CONCEPT <name> <concept-expression>)        % new concept
        (DEFINE-ROLE <name>)                                % new role
        (CREATE-IND <name> <individual-expression>)         % new individual
        (IND-ADD <name> <individual-expression>)            % add information
        (ADD-RULE <concept-name> <concept-expression>)      % add a rule
```

Fig. 2. The Original CLASSIC Knowledge Base Grammar

all in the context of a simple, learnable clean frame system.[5] In this respect, the work was worth publishing; but in retrospect it was naive of us to think that we could "just" build it and use it.

3 The Transition to Practice

Given the simplicity of the original design of CLASSIC, we held the traditional opinion that there was essentially no research left in implementing the system and having users use it in applications. In late 1988, we concluded a typical AI programming effort, building a CLASSIC prototype in COMMON LISP. As this version was nearing completion, we began to confer with colleagues in an AT&T development organization about the potential distribution of CLASSIC within the company. Despite the availability of a number of AI tools, an internal implementation of CLASSIC held many advantages: we could maintain it and extend it ourselves, in particular, tuning it to our users; we could assure that it integrated with existing, non-AI environments—our many legacy systems; and we could assure that the system had a well-understood, formal foundation (in contrast to virtually all AI tools commercially available at the time). Thus we initiated a collaborative effort to create a truly practical version of CLASSIC, written in C. Our intention was to develop the C version, maintain it, create a training course, and eventually find ways to make it usable even by novices. Meanwhile, as the C effort progressed, we began to experiment with our first applications using the LISP prototype.

The issues and insights reported in the next several sections arose over an extended period of time, in which we collaborated on the design of the C version of CLASSIC, and developed several substantial and different types of applications, and, later, produced a third version of CLASSIC (in C++). Two of the applications were more or less along the lines we expected. But the others were not originally foreseen. The first two applications—a software information system using classification to do retrieval [18], and what could be considered a "semantic data model" front end to a relational database of program cross-reference

[5] Some, though not all, of these features were independently introduced in the other two description logic-based systems being developed at about the same time—LOOM [24] and BACK [39].

information [42]—were planned from the beginning. Other types of applications were unanticipated. One family of applications (FindUR [26]) used CLASSIC to manage background ontologies, which were used to do query expansion in World-Wide Web searches. The description logic system organized descriptions and detected problems in collaboratively generated ontologies maintained by people not trained in knowledge representation. Another application proved to be the basis of a commercially deployed NCR product. It was the front end for data analysis in a data mining setting and was marketed as the "Management Discovery Tool" (MDT). Our family of configuration applications [44,31,30,28] was the most successful of the applications. Fifteen different configurators were built using CLASSIC, and these have gone on to have processed more than $6 billion worth of orders for telecommunications equipment in AT&T and Lucent Technologies.

In addressing the needs of these applications, we developed interim versions of the LISP CLASSIC system and received feedback from users along the way. Thus the "transition research" reported here was stimulated both by the need to construct a usable system in general and by the demands of real users in real applications.

4 Feedback from Practice

The process of implementing, re-implementing in C, and having application builders use CLASSIC and provide us with feedback, resulted in significant changes. Some of these arose because of simple facts of life of providing real software to real people, some arose because it is impossible to anticipate many key needs before people start to use a system, and some arose because it is hard to truly complete the formal analysis before you begin building a system.[6]

There are many significant lessons to be learned from the attempt to move an abstract KR service into the real world, some of which are sociological and very general, and some of which are very low-level and relate only to implementation details. We restrict ourselves here mainly to technical considerations that influenced the ultimate shape of the CLASSIC language and KB operations, and critical differences between the original proposal and the final form. We look at a small number of examples in each case. Even looking at the feedback from practice to theory involving only basic language and KB operations, there are numerous factors that end up shaping the final version of a logic. Here we cover issues relating to software system development, *per se*; implementation considerations; needs of users; the needs of specific applications; and the demand from practice that all details be worked out.

[6] This is by no means to say that such prior formal analyses are not also critical to the success of a project. Here, our intention is simply to focus on the relatively unheralded role that implementation and use play in the success and ultimate shape of KR systems.

4.1 Creating and Supporting a System

Even if the setting is not a commercial one, the intent to create and release a system carries certain obligations. In the case of CLASSIC, our development collaborators made one thing very clear: do not release software of which you are unsure. In particular, it is important not to include features in an initial release that you might choose later to remove from the language. Once someone starts to use your system, it is almost unpardonable to release a future generation in which features that used to be supported no longer are (at least with respect to the central representation language constructs).

Even in a research setting, while we are used to playing with features of a language to see what works best, once software is created, it is awkward at best to nonmonotonically change the supported language. In CLASSIC, this meant that we needed to carefully consider every constructor, to make sure that we were confident in its meaning and utility. In particular, we were forced by our commitment to efficient implementation to exclude the original, general **SAME-AS** constructor from the initial system specification because its implementation was appearing increasingly more complex (see Section 4.2).

This constraint also forced us to abandon a constructor we were considering that would allow the expression of concepts like "at least one female child" (this type of constuct has come to be known as a "qualified number restriction"). In the general case, such a constructor rendered inference intractable [34] and we wanted to avoid known intractability. Had our users demanded these constructors, we might have tried for a reasonable partial implementation. Unfortunately, we did not have a handle on an incomplete algorithm that we could guarantee would only get more complete with subsequent releases, and which had a simple and understandable description of what it computed. The latter is particularly important, since otherwise users might expect certain conclusions to be drawn when the algorithm in fact would miss them. Such mismatched expectations could be disastrous for the acceptability of the product. We were strongly told that it was better to eliminate a construct than to have a confusing and unintuitive partial implementation of it. Upward compatibility dictated that even with incomplete algorithms, subsequent versions of the system would still make at least all inferences made in earlier versions. Thus, we were better off from this perspective in keeping the language simple, and left this construct out of the initial released version.

There were other influences on the evolution of CLASSIC because of this prosaic, but important type of constraint. For a while, we had contemplated a role inverse construct (see Section 4.2). For example, as was common in many older semantic network schemes, we wanted to say that `parent` was the inverse of `child`. While we had not designed such a construct into the original specification, it appeared to be potentially very useful in our applications. We worked out several solutions to the problem, including a fairly obvious and general one that allowed inverses to be used for any role at any time. However, the cost appeared to be very high, and it was not even clear that we could implement the most general approach. As a result, we were forced to abandon any attempt to implement role inverses in the first released version of CLASSIC. We were not

totally confident that we could stick with any initial attempt through all subsequent releases, and we did not want to start with an awkward compromise that might be abandoned later.

Another consideration is harder to be technical about or to quantify, but was equally important to us. As the system began to be used, it became clear that we needed to be as sure as possible that our constructors were the best way to carve up the terminological space. In description logics, there are many variant ways to achieve the same effect, some of which are more elegant and comprehensible than others. Since we intended to put our software in the hands of non-experts, getting the abstraction right was paramount. Otherwise, either the system would simply not be used, or a second release with better constructors would fail to be upward compatible. In CLASSIC's case, we put a great deal of effort into an assessment of which constructors had worked well and which had not in previous systems.

Finally, as mentioned, CLASSIC early on became a coordinated effort between research and development organizations. Once we had the development (C language) version of CLASSIC released and a set of commercial applications in place, we thought that we had reached a steady state. The research team would experiment with adding new features first to the LISP version of CLASSIC, and then the development team would port those features to the C version. Both teams were involved in both activities, so that while the C version of CLASSIC would lag behind the LISP version, it would never be too far behind.

Unfortunately, this anticipated mode of operation turned out to have several problems. First, for pragmatic reasons in getting off the ground, the C version never did have all the features of even the initial LISP version. Second, once the C version was suitable for the commercial applications, there was no short-term development reason for adding features to it. Additions to the C version would be supported by development resources only in response to needs from current or proposed applications. Third, the research team had neither the resources nor, indeed, the expertise to change the C version.

These mundane human resource constraints meant that it was very unlikely that the C version of CLASSIC would ever approach the capabilities of the LISP version. Once we realized this, we decided that the only solution would be to create a combined version of CLASSIC in a language acceptable for both research and development. This combined version, which we called NEOCLASSIC, was written in C++, the only language acceptable to development that was reasonable for research purposes. NEOCLASSIC was designed to immediately supplant the C version of CLASSIC and was supposed to quickly acquire the features of the LISP version, modified as appropriate. Implementation of NEOCLASSIC proceeded to the point that it was usable and as featureful as the C version, but ultimately corporate changes and personnel issues caused the work to be scaled back.

4.2 Implementation Considerations

There were at least two aspects of the implementation effort itself that ultimately ended up influencing the language and operation design. One was the sheer

difficulty of implementing certain inferences, and the other was the normal kind of tradeoff one makes between time and space.

Complex Operations. We began our implementation by attempting to construct a complex subsumption algorithm that included **SAME-AS** relations between roles. For example, we wanted to be able to detect that a concept that included

```
(AND (SAME-AS (boss) (tennis-partner))
     (SAME-AS (tennis-partner) (advisor)))
```

was subsumed by (i.e., was more specific than) a concept that included

```
(SAME-AS (boss) (advisor));
```

in other words, that someone whose boss is her tennis partner, and whose tennis partner is her advisor, must of necessity be someone whose boss is her advisor.

Because **SAME-AS** could take arbitrary role *paths*, roles could possibly be unfilled for certain individuals, and roles could have more than one filler; this made for some very complex computations in support of subsumption. Cases had to be split, for example, into those where roles had some fillers and those where they had none, and less-than-obvious multiplications and divisions had to be made in the presence of number restrictions. More than once, as soon as we thought we had all cases covered, we would discover a new, more subtle one that was not covered. When we finally came to a point where we needed a result from group theory and the use of factorial to get some cardinalities right, we decided to abandon the implementation.

As it turns out, our attempts at a straightforward and efficient solution to what appeared to us to be a tractable problem were thwarted for a deep reason: full equality for roles is undecidable. This result was later proved by Schmidt-Schauss [41] and Patel-Schneider [37]. Thus, our implementation enterprise could *never* have fully succeeded, although we did not know it at the time. The end result of all of this was an important change to the CLASSIC language (and therefore needed to be reflected in the semantics): we moved to distinguish between *attributes*, which could have exactly one filler, and other roles, which could have more than one filler. **SAME-AS** could then be implemented efficiently for attributes (using ideas from [1]), appropriately restricted to reflect the dichotomy. In the end, the distinction between attributes and multiply-filled roles was a natural one, given the distinction between functions and relations in first-order logic, and the common use of single-valued relations in feature logics and relational databases.[7]

Other aspects of CLASSIC evolved for similar reasons. For example, while our original specification for a **TEST** concept was conceptually sufficient, we left it

[7] Attributes correspond to functionally-dependent columns in relations, whereas multiply-filled roles would most easily correspond to two-column relations. These correspondences turned out to be of great use to us when we subsequently attempted to interface CLASSIC to a relational database [14].

to the user to specify (optionally) a "realm" (i.e., *CLASSIC* or *HOST*). This made the processing more complex—and different—for concepts that had **TEST**s than for those that did not. It was also the only case of a concept construct for which the user had to specify a realm at all. In order to make the code more simple and reliable, and the interface more uniform, we eventually substituted for **TEST** two different constructors (**TEST-C**, **TEST-H**), which would each be unambiguous about its realm.

Implementation Tradeoffs. It is well known that creating computer programs involves making tradeoffs between time and the use of space. In most cases, decisions made because of efficiency should not be of consequence to the system's functional interface. However, some tradeoffs can be extremely consequential, and yet never occur to designers of an unimplemented language.

One tradeoff that affected our user language involved the form and utility of role inverses. If one could afford to keep backpointers from every filler back to every role it fills, then one could have an (**INVERSE** <role>) construct appear any place in the language that a role can. This would be quite convenient for the user and would contribute to the uniformity of the language—but it would also entail significant overhead. An alternate view is to force users to explicitly declare in advance any role inverses that they intend to use at the same time the primitive roles are declared. Then, backpointers would be maintained only for explicitly declared roles. The point here is not which approach is best, but rather that practical considerations can have significant effects on a pure language that never takes such things into account. Given the large difference between the two approaches, and inferential difficulty that results from including inverses in the language, we chose to exclude them altogether from the first release. A more recent version of CLASSIC included them, since we subsequently had a chance to think hard about the interplay between language, operation, and implementation design.

While the original specification of CLASSIC did not account for *retraction* of information, our applications soon forced us into providing such a facility. In this case, retraction became one of the key reflectors of implementation tradeoffs. Most reasonable implementations of retraction in inference systems keep track of dependencies. Given extremely large knowledge bases, it may be too expensive (both in space and time) to keep track of such dependencies at a very fine-grained level of detail. Because of this, CLASSIC has a unique medium-grain-sized dependency mechanism, such that for each individual, all other individuals can be found where a change to one of them could imply a change to the original. This medium-grained approach saves space over approaches (e.g., [24]) that keep track of dependencies at the assertion level, or approaches that keep track of all dependencies in a truth-maintenance fashion. The reduction in the amount of record-keeping also saves time, which we believe even results in an overall faster system.

4.3 Serving the General User Population

Real use of a system, regardless of the particular applications supported, immediately makes rigorous demands on a formalism that may otherwise look good on

paper. We consider three types of issues here: (1) comprehension and usability of the language by users; (2) specific features of the user interface, e.g., error-handling and explanation, that make the system more usable; and (3) getting around gaps in the logic.

Usability of the Language. Concepts like "usability" are admittedly vague, but it is clear that users will not stick with a system if the abstractions behind its logic and interface do not make sense. Formal semantics makes precise what things mean, and it behooves us to provide such formal bases for our logics. However, how simple a language is to learn and how easy it is to mentally generate the name of a function that is needed are more likely the *real* dictators of ultimate success or failure.

In the CLASSIC world, this meant (among other things) that the language should be as uniform as possible—the more special cases, the more problems. Just being forced to think about this led us to an insight that made the language better: there was in general no good reason to distinguish between what one could say about an individual and what one could use as part of a concept. (Note in the grammar of Figure 1 that concept-expressions and individual-expressions are treated differently.) The **FILLS** constructor should have been equally applicable to both; in other words, it makes sense to form the general concept of "an automobile whose manufacturer is Volvo," where Volvo is an individual:

```
(AND AUTOMOBILE (FILLS manufacturer Volvo))
```

In the original specification, we thought of role-filling as something one does exclusively in individuals. The one sticking point to a generalization was the **CLOSE** constructor, which we felt did not make much sense for concepts; but as we see below, further thinking about **CLOSE** (instigated by user concerns) eventually led us to determine that it was mistakenly in the language in the first place. As a result, the types of descriptions allowable as definitions of concepts and for assertions about individuals could be merged.

There were other simplifications based on generic user concerns like understandability that helped us derive an even cleaner logic. For example, the **PRIMITIVE** and **DISJOINT-PRIMITIVE** concept-forming constructors,

which had a firm semantics but were found problematic by non-experts in actual use, were removed from the language and better instantiated as variants on the concept-defining interface function. The conceptually adequate but awkward arguments to **DISJOINT-PRIMITIVE** were also simplified.

While we provided all of our research papers, potential users demanded usage guidelines aimed at non-PhD researchers, to aid their comprehension of the logic. In an effort to educate people on when a description logic-based system might be useful, what its limitations were, and how one might go about using one in a simple application, a long paper was written with a running (executable) example on how to use the system [12]. This paper discussed typical knowledge bases, useful "tricks of the trade," ideas in our logic that would be difficult for non-KR people, and a conventional methodology for building CLASSIC KB's.

Motivated by the need to help users understand CLASSIC's reasoning paradigm and by the need to have a quick prototyping environment for showing off novel functionality, we developed several demonstration systems. The first such system was a simple application that captured "typical" reasoning patterns in an accessible domain—advising the selection of wines with meals. While this application was appropriate for many students, an application more closely resembling commercial applications in configuration was needed to give more meaningful demonstrations internally and to provide concrete suggestions of new functionality that developers might consider using in their applications. This led to a more complex application concerning stereo system configuration, which had a fairly elaborate graphical interface [29,28]. Both of these applications have subsequently been adapted for the Web.

Motivated by the need to grow a larger community of people trained in knowledge representation in general and description logics in particular, we collaborated with a corporate training center to generate a course. Independently, at least one university developed a similar course and a set of five running assignments to help students gain experience using the system. We collaborated with University of Pittsburgh on the tutorial to support the educators and to gather feedback from the students. The student feedback from yearly course offerings drove many of our environmental enhancements such as enhanced explanation support for contradictions, pruning, and debugging.

All of this effort in building user aids seemed truly "ancillary" at the beginning, but proved to be crucial in the end.

Human Interface Features. Even with a perfectly understandable and intuitive logic, a minimal, raw implementation will be almost impossible to use. In general, our customers told us, the system's development environment (for building and debugging knowledge bases) was a make-or-break concern. For example, logics discussed in papers do not deal with issues like *error-handling*, yet real users can not use systems unless they get meaningful error-reporting and reasonable error-handling, especially when the KR system is embedded in a larger system. As a result of direct and strong feedback from users, the released version of CLASSIC had extensive error-handling, including well-documented return codes and rational and consistent error returns.

More specifically, our configuration applications relied heavily on the detection of *contradictions*, since users would, among other things, try out potential updates in a "what-if" mode. Certain input specifications might lead to an inconsistency with the updates that had previously been made. One of the key aspects of contradiction-handling, then, was the need to roll back the effects of the update that caused such an inconsistency in the knowledge base. Since CLASSIC was able to do elaborate propagation and rule-application, a long and ramified inference chain may have been triggered before a contradiction was encountered, and unless every piece of that chain were removed, the knowledge base would be left in an incoherent state. This need led us to consider ways to unravel inferences, including the possible use of a database-style "commit"

operation (i.e., the knowledge base would never be changed until all inferences concluded successfully).

We eventually settled on a more conventional AI approach using dependencies, which gave us a general facility that not only would guarantee the KB to be returned to a meaningful state after a contradiction occurred, but would allow the user direct *retraction* of facts previously told. As it turned out, the availability of such a retraction capability was critical in "selling" the application to its sponsors, since the ability to explore alternative options in unconstrained ways was essential to the interactive customer sales process.

Another generic area that needed attention was *explanation* of reasoning—a topic relatively ignored by the KR community. If users are to build nontrivial KB's, they will need help in understanding and debugging them; they will need to know why an inference failed, or why a conclusion was reached. While the expert systems community may have learned this lesson, it is an important one for those working in general KR as well. Our users made a very strong case to us that such a feature was critical to their successful construction of knowledge bases.

We responded by adding an explanation mechanism to CLASSIC [25,27]. Since the key inference in CLASSIC is subsumption, its explanation forms the foundation of an explanation module. Although subsumption is calculated procedurally in CLASSIC, we found it necessary to provide a declarative presentation of CLASSIC's deductions in order to reduce the length of explanations and to remove the artifacts of the procedural implementation. We used an incremental proof-theoretic foundation and applied it to all of the inferences in CLASSIC, including the inferences for handling constraint propagation and other individual inferences. This basic explanation foundation has proved useful and general and since then has been used (in joint work with Ian Horrocks and Enrico Franconi) as the foundation for a design for explaining the reasoning in tableaux-based description logic reasoners, and also (in joint work with James Rice) in an implemented system for explaining the reasoning in a model-elimination theorem prover at Stanford.

As soon as we had both explanation and appropriate handling of contradictions in CLASSIC, we found that specialized support for explanation of contradictions was called for. If an explanation system is already implemented, then explaining contradictions is *almost* a special case of explaining any inference, but with a twist. Information added to one object in the knowledge base may cause another object to become inconsistent. Typical description logic systems, including CLASSIC, require consistent knowledge bases, thus whenever they discover a contradiction, they use some form of truth maintenance to revert to a consistent state of knowledge (as mentioned above), removing conclusions that depend on the information removed from the knowledge base. But a simple-minded explanation based solely on information that is currently in the knowledge base would not be able to refer to these removed conclusions. Thus, any explanation system capable of explaining contradictions would need to access its inconsistent states as well as the current state of the knowledge base.

Another issue relevant to explanation is the potential incompleteness of the reasoner. In particular, a user might have an intuition that some conclusion

should have been reached, but the system did not reach it. To explain this might in general require using a different, complete reasoner, but frequently occuring special cases can be built into the system itself.[8]

As CLASSIC makes it easy to generate and reason with complicated objects, our users found naive object presentations to be overwhelming. For example, in our stereo demonstration application, a typical stereo system description generated four pages of printout. This contained clearly meaningful information, such as price ranges and model numbers, but also descriptions of where the component might be displayed in the rack and which superconcepts were related to the object. In fact, in some contexts it might be desirable to print just model numbers, while in other contexts it might be desirable to print price ranges and model numbers of components.

To reduce the amount of information presented in CLASSIC explanations we added facilities for describing what is interesting to print or explain on a concept-by-concept basis. This led us to a meta-language for matching "interesting" aspects of descriptions [25,8]. The approach provides support for encoding both domain-independent and domain-dependent information to be used along with context to determine what information to print or explain. The meta-language essentially extends the base description logic with some carefully chosen auto-epistemic constructors ("Is at least one filler known?") to help decide what to print. As a result, in one application object presentations and explanations were reduced by an order of magnitude, which was essential in making the project practical.

Overcoming Gaps in the Logic. Another key point of tension between theory and practice is the notion of an *"escape"* for the user, e.g., a means to get around an expressive limitation in the logic by resorting to raw LISP or C code. As mentioned above, we included in the original specification a **TEST** construct, which allowed the user to resort to code to express sufficiency conditions for a concept. In the original paper, we did not technically include **TEST**s in concept definitions, since no formal semantics was available for it. We quickly provided guidelines (e.g., avoiding side-effects) that could guarantee that **TEST**-defined concepts could fit into our formal semantics, even if the **TEST** code itself was opaque to CLASSIC. But our view was that the **TEST** construct was not intended to be a general programming interface.

As it turned out, **TEST**-concepts were one of the absolute keys to successful use of CLASSIC. In fact, they not only turned out to be a critical feature to our users, but as we observed the patterns of tests that were written in real applications, we were able to ascertain a small number of new features that were missing from the language but fundamental to our applications. First, we discovered that users consistently used **TEST**s to encode simple numerical range restrictions; as we mention below, this led us to create **MAX** and **MIN** constructors for our concept language. Later, in one significantly large and real-world application,

[8] In the case of CLASSIC, inferences not supported by the modified semantics of individuals used in subsumption reasoning fall into this category.

we found only six different patterns of **TEST** concepts, with over 85% of these falling into only two types; one was computing via a function a universal restriction on a role (actually, a numerical range), and the other was computing a numerical filler for a role (a simple sum). We have subsequently made additions to CLASSIC to accommodate these common patterns of usage (i.e., "computed rules"), and have found that newer versions of the same knowledge base are substantially simpler, and less prone to error (the original **TEST**s were written to achieve some of their effects by side-effect, which we subsequently eliminated).

Thus, while our original fear was that an escape to LISP or C was an embarrassing concession to implementation, and one that would destroy the semantics of the logic if used, our **TEST**s were never used for arbitrary, destructive computation. Rather, this mechanism turned out to be a means for us to measure specifically where our original design was falling short, all the while staying within a reasonable formal semantics.[9]

4.4 Meeting the Needs of Particular Applications

As soon as a system is put to any real use, mismatches or inadequacies in support of particular applications become very evident. In this respect, there seems to be all the difference in the world between the few small examples given in research papers and the details of real, sizable knowledge bases. As mentioned, we took on several significant and different types of applications. While the demands from each of them were somewhat different, they clearly did *not* demand that we immediately extend CLASSIC to handle the expressive power of full first-order logic. In fact, the limited number of extensions and changes that arose from the interaction with individual applications are useful in all of them, and all stay within the original spirit of simplicity.

Language and KB Operation Features. Among the first needs we had to address was the significance of numbers and strings. Virtually all of the applications needed to express concepts limiting the values of roles that had *HOST* values in them, as in, for example, "a manager whose salary is between 20000 and 30000." On the one hand, this need vindicated our original decision to integrate host information in a serious manner.[10] On the other, as mentioned above, the need to create **TEST**-concepts just to test simple ranges like this showed us that we would have a hard time measuring up to almost any application that

[9] As evidence of the continuing general lack of appreciation of the more theoretically-inclined towards pragmatic issues, consider that one of the reviewers of our 1989 paper [5] called **TEST**s "an abomination." Yet, not only were they undeniably critical to our users, we managed to keep them in line semantically and they provided concrete input concerning the expressive extensions that were most required by our users.

[10] We should point out that integration here is not just a simple matter of allowing numbers or strings in roles; it has ramifications for the language syntax and parsing, part of the concept hierarchy must be built automatically, data structures for *CLASSIC* individuals need to be carefully distinguished from arbitrary LISP structures, etc.

used real data (especially if it came from a DBMS). Thus, recent versions of CLASSIC have new concept types that represent ranges of *HOST* values.[11] These are integrated in a uniform manner with other concept constructors, and the semantics accounts for them.

Another major consequence of dealing with a significant application is the reality of *querying* the KB. Our original design of CLASSIC (as was the case with virtually all frame systems) paid scant attention to queries other than those of the obvious sort, e.g., retrieving instances of concepts. Once we began to see CLASSIC as a kind of deductive database manager, we were forced to face the problem that our querying facilities were very weak. This led to the design of a substantial query language for CLASSIC, which could handle the needed object-oriented queries, as well as the SQL-style retrievals that are so common in the real world of information management. While this is not profound (although the query language we developed has some novel features and is itself an important contribution), the key point is that it was the attempt at application that made us realize that an entire critical component was missing from our work.

Two other consequences of this sort bear brief mention.

First, our simple notion of a **TEST** was sufficient to get us off the ground. Our intention was to pass the individual being tested to the test function as a single argument. As it turned out, our users needed to pass other items in as arguments. For example, if the test were a simple function to compute whether the value of a role were greater than some number, say 5, then the number 5 should have been coded directly into the test function; this, in turn, would have led to the creation of many almost-identical functions—unless we provided the ability to pass in additional arguments. We have done so in the latest versions of CLASSIC.

Second, our original design of rules was a sufficient foundation, but it required a named concept to exist as the left-hand-side of the rule. As soon as some of our users tried to use this, they found that they had to construct concepts artificially, just to serve to invoke the rules. While this posed no conceptual problem for the logic, and no semantic aberration, it became a practical nightmare. Thus, it was important to extend our rules to allow a filter; in other words, the rule could be associated with the most general named concept for which it made sense, but only fired when a filtering subcondition was satisfied. This now avoids needless creation of artificial concepts.

API's. Finally, an important consideration was the relationship between our KR system and the application that used it. In the vast majority of our applications, CLASSIC had to serve as a tightly integrated component of a much larger overall system. For this to be workable, CLASSIC had to provide a full-featured application programming interface (API) for use by the rest of the system.

[11] **MAX** and **MIN** have instances that are numbers; e.g., `(MAX 25)` represents the set of integers that are less than or equal to 25. These are used to restrict the *value* of a filler of a role; for example, we could use **MAX** to specify the value restriction on a person's age, as in `(AND PERSON (ALL age (MAX 25)))`. **AT-LEAST** and **AT-MOST**, on the other hand, restrict the *number of fillers* of a role, not their values.

Our most complete API was in the NeoClassic (C++) system. It had the usual calls to add and retract knowledge and to query for the presence of particular knowledge. In addition, there was a broader interface that let the rest of the system receive and process the data structures used inside NeoClassic to represent knowledge, but without allowing these structures to be modified outside of NeoClassic.[12] This interface allowed for much faster access to the knowledge stored by NeoClassic, as many accesses were simply to retrieve fields from a data structure. Further, direct access to data structures allowed the rest of the system to keep track of knowledge from NeoClassic without having to keep track of a "name" for the knowledge, and also supported explanation.

A less-traditional interface that is provided by both lisp classic and NeoClassic is a notification mechanism ("hooks"). This mechanism allows programmers to write functions that are called when particular changes are made in the knowledge stored in the system or when the system infers new knowledge from other knowledge. Hooks for the retraction of knowledge from the system are also provided. These hooks allow, among other things, the creation of a graphical user interface that mirrors (some portion or view of) the knowledge stored in the representation system.

Lately, others in the knowledge representation community have recognized the need for common API's, (e.g., the general frame protocol [17] and the open knowledge base connectivity [16]) and translators exist between the general frame protocol API specification and classic.

4.5 Revisiting What Looked Good on Paper

Probably more commonly than researchers would like to admit, theoretical KR papers are not always what they seem. While theorems and formal semantics help us get a handle on the consequences of our formalisms, they do not always do a complete job; it is also not unheard of for them to contain mistakes. Our own experience was that several parts of our original formalism were clarified substantially by the experience of having to implement an inference algorithm and have it used on real problems. In each case, a change was necessary to the original formal work to accommodate the new findings. Because of the complexities and subtleties of real-world problems, and the extreme difficulty of anticipating in the abstract what real users will want, it seems that this type of effect is inevitable, and a critical contribution of practice over pure "theory."

For example, we had originally proposed that **CLOSE** could appear in a description applied to an individual, to signal that the role fillers asserted by the description were the only fillers. Thus, one could assert of Dale,

```
(AND (FILLS friend Audrey)
     (FILLS friend Harry)
     (CLOSE friend));
```

[12] Of course, as C++ does not have an inviolable type system, there are mechanisms to modify these structures. It is just that any well-typed access cannot.

this was to mean that Audrey and Harry were Dale's *only* friends. We thought of this, semantically, as a simple predicate closure operation. However, once a real knowledge base was constructed, and users started interacting with the system, we discovered a subtle ordering dependency: pairs of **CLOSE** constructors could produce different effects if their order were reversed; this occured because the first closing could trigger a rule firing, whose effect could then be to enable or block another rule firing in conjunction with the second closing. This led us to discover that our original characterization of **CLOSE** was in general wrong. In reality, it had an autoepistemic aspect, and thus closing roles had to become an operation on an entire knowledge base, and could not be part of a larger expression. **CLOSE** was thus removed from the description language and made a knowledge base operation.

We had a small number of similar experiences with other aspects of the language. For instance, our original estimation was that a certain part of CLASSIC's reasoning with individuals was complete for the purposes of subsumption checking. In implementation, we had to look substantially closer at what properties of individuals could count in the subsumption of concepts (individuals could appear in concepts originally in **ONE-OF** constructs, and later, in **FILLS** expressions). In doing so, we discovered that while the implementation actually did what we thought was right—it ignored contingent properties of individuals in subsumption calculations—the semantics was wrong. We eventually found a suitable, interesting, and somewhat non-standard semantic account that described what we *really* meant [7]. Moreover, it was discovered ([40] and, independently, [7]) that reasoning with **ONE-OF** according to standard semantics is intractable. In retrospect, our belief is that it would have been a mistake to omit individuals from concept descriptions because of this complexity result, and that our intuitions serendipitously led us to a good compromise. So, while formal semantics are a good foundation on which to build, they are not necessarily what the designers mean or what the users need to understand.

Being forced by implementation to get every last detail right also caused us ultimately to better understand our **TEST** constructs. Given when they would be invoked, it eventually became clear (thanks to a key user's discovery) that **TEST**s, which were originally two-valued functions, had to be *three*-valued: since CLASSIC supports partial information about individuals, it is possible for a test to "fail" at one point and succeed later, even with strictly monotonic additions to the KB. If the test truly failed the first time, and the individual were determined *not* to satisfy a description based on this failure, nonmonotonicity would be introduced in an inappropriate way. Thus **TEST** functions need to tell their caller if the individual provably satisfies the test, provably fails it, or neither.

4.6 Other Important Influences

While our focus here has been on the feedback from our practice with CLASSIC to our theory, it is important to point out that our practical work on CLASSIC both spawned and benefited from other theoretical work.

For example, we had not worried about that fact that expanding the definitions of concepts could, in the worst case, lead to an exponential blow-up in the space required to represent a CLASSIC knowledge base, but we did not know whether there was perhaps a better method. Work by Nebel [34] showing that there is an inherent intractability in the processing of description logic knowledge bases made the existence of such a method unlikely.

As mentioned, the original **CLOSE** constructor had to be abandoned because of implementation ordering difficulties. We replaced the **CLOSE** constructor with an operation on knowledge bases, to which we gave an operational semantics. Donini, *et al.*, [20] realized the true epistemic nature of this operation, and pointed out that the trigger rules used in CLASSIC and LOOM also have such an epistemic nature. As a result, a theory of epistemic DL's was developed, where rules like `PERSON` $\rightsquigarrow$`(ALL parents PERSON)`, hitherto given an operational interpretation, were integrated into the knowledge base by using a modal operator **K**: **K**`(PERSON)` $\Longrightarrow$ `(ALL parents PERSON)`, and thus given a denotational semantics. This semantics provides a justification for the operational treatment provided in CLASSIC.

The successful implementation and use of filtering for elimination of uninteresting information about individuals led to its formalization through the notion of *patterns*—descriptions with variables occurring in some places instead of identifiers—and pattern matching [8]. This formal work has been extended to other languages [3], and may have applications to knowledge-base integration, where concepts from one ontology may be matched against corresponding parts of the other ontology, in order to discover commonalities [6].

Inspired by the need for both domain-independent extensions (e.g., qualified number restriction), and domain-specific ones (e.g., reasoning with dates, plans, etc.), the CLASSIC implementation was analyzed, rationalized and generalized to an architecture that supports the addition of new concept constructors [4]. In fact, the last version of LISP CLASSIC (release 2.3) has features for adding subsumption reasoning for some **TEST**-concepts, because a description like `(TEST-H initialSubstring "reason")`—denoting strings that begin with the letters `r-e-a-s-o-n`—can be viewed as just a syntactic variant of the description `(INITIAL-SUBSTRING "reason")`, which makes it clear that a new constructor is being used. Note that the addition of arguments to **TEST**-concept functions was crucial in this step to extensibility.

5 Modern CLASSIC

The result of the long and arduous trail implied above, from typical research paper to practical system, was a significant improvement in the CLASSIC language and the clarity of operations on a CLASSIC knowledge base. The basic expression grammar was simplified and made more uniform (see Figure 3), and the semantics was adjusted to be truer to our original intention. KB operations were streamlined and made more useful, and error-handling and retraction were added. The resulting system is unarguably superior to the original in every way:

```
<concept-expression> ::=
      THING | CLASSIC-THING | HOST-THING | NUMBER | STRING |
      <concept-name> |
      (AND <concept-expression>+) |
      (ALL <role-name><concept-expression>) |
      (AT-LEAST <positive-integer><role-name>) |
      (AT-MOST <non-negative-integer><role-name>) |
      (FILLS <role-name> <individual-name>+) |          % added for uniformity
      (SAME-AS (<attribute-name>+) (<attribute-name>+)) |     % restricted
      (TEST-C <function><arg>*) |       % clarified; arguments added; 3-valued
      (TEST-H <function><arg>*) |       % clarified; arguments added; 3-valued
      (ONE-OF <individual-name>+) |
      (MAX <number>) |                                        % added
      (MIN <number>)                                          % added
<individual-expression> ::=
      <concept-expression> |                % made uniform with concepts
      <individual-name> |
      <host-language constant>
```

Fig. 3. The Resulting CLASSIC Concept Language

it has new constructs that meet real needs, substantial parts of it have been validated by use, the overall interface makes more sense, it is cleaner and more elegant, and it is devoid of flaws that were subtly hidden in the original.

The effects of the pragmatic factors we have described here are varied, and not easily classified. But they are clearly substantial and were critical to the success and ultimate form of CLASSIC. To summarize, here are some of the most important changes that were driven by the attempt to "reduce" the system to practice and put it to the test of real use:

- *language improvements:* equal descriptive power for individuals and concepts; distinction between attributes and multiply-filled roles; **SAME-AS** applicable to attributes only and efficiently computable; arguments for **TEST**-concepts; three-valued **TEST**s; completely compositional language with no order dependencies; numeric range concepts; rules with filter conditions, no longer requiring artificial concepts; realms of **TEST**-concepts unambiguous and **TEST** constructs made uniform with other parts of language; computed rules;
- *interface improvements:* primitive and disjoint primitive definition as KB operators; disjoint primitive specification simplified; **CLOSE** as a KB operator; sophisticated query language and implemented query processor; complete API for embedded use;
- *system features:* comprehensive error-reporting and handling; extensive explanation capabilities; filtering language for pruning; renaming of concepts; retraction of "told" information; contradiction-handling.

Finally, we completed the cycle by embarking on an actual formal proof of the tractability of CLASSIC and the completeness of our reasoner [7]. This proof

was more difficult than usual because the language lacks negation, so standard techniques could not be applied. We ended up using an abstraction of the implementation data structure for the proof, and we must admit that it took the trained eye of a very devoted and skilled reviewer to get the details right. So, while it would have been nice to have come up with this proof before we even proposed the logic and tried to implement it, it is very doubtful that we would have succeeded, without the experience of practice to guide us.

6 Lessons

The main lesson to be learned here is that despite the ability to publish theoretical accounts of logics and their properties, the true theoretical work on KR systems is not really done until issues of implementation and use are addressed head-on. The basic ideas can hold up reasonably well in the transition from paper to system, but traditional research papers miss many make-or-break issues that determine a proposal's true value in the end. Arguments about needed expressive power, the impact of complexity results, the naturalness and utility of language constructs, etc., are all relatively hollow until made concrete with specific applications and implementation considerations.

Although a complete formal specification of a knowledge representation system (including an algorithmic specification of the inferences that the system is required to perform and a computational analysis of these inferences) is essential, the presence of a formal account is not sufficient for the success of the system. There is no guarantee, for example, that a formally tractable knowledge representation system can be effectively implemented, as it may be exceedingly difficult to code the inference algorithms or other portions of the system efficiently enough for use or perspicuously enough to tell if they are correct. Even then, there is no guarantee that the resulting system will be useful in practice, even if it appears at first glance to meet some apparent needs. Finally, getting the formal specification *really* right is an extremely difficult task, especially for systems that perform partial reasoning or which have non-standard but useful constructs. All told, *the implementation and use of the system is a vital complement to work on knowledge representation "theory."* It can illuminate problems in the formal specification, and will inevitably provide real problems for the theory side to explain.

Our experience with CLASSIC has taught us this lesson in some very specific ways. Any hope of having the system make a real impact (e.g., in a product) rested on some very practical considerations that in some cases were impossible to anticipate before interacting with developers. We learned through extensive interaction with our developers that issues like upward compatibility and simplicity were in some ways much more important than individual features. We learned that usability issues such as explanation, contradiction-handling, and pruning were critical to longevity and maintenance in applications and were much more important than additional language constructs. We learned that attention to complexity (although not maniacal concern with it) was very much

worth the effort, because of the critical impact of performance and predictability on acceptance of the system. We also learned that we could not afford to be totally rigid on any point—be it language features, complexity, or names of functions—without jeopardizing potential use of the system. The feeling of the CLASSIC group is that the resulting system is clearly far better than anything we could have built in a research vacuum. And the effort of reducing our ideas to a practical system generated a great deal of research—on language constructs, complexity, and even formal semantics—that was not only interesting, but important simply by virtue of the very fact that it arose out of real problems.

At a more strategic level, one very important lesson for us was the significance of a certain kind of conservatism. We could have invested a large amount of time designing features and providing expressive power (and implementation complexity) that would have, as it turned out, gone completely to waste. On the flip side, our users gave us clear and direct evidence of features that they did need, and that we were not providing, via our **TEST** construct, which, to be honest, surprised us both in its criticality and in the simple and regular ways in which it was used—not to mention the smallness of the number of needed extensions. All told, our decision to start with a small (but not hopelessly impoverished) language, with room for growth in a reasoned fashion, was clearly a successful one. While such emphasis on simplicity might not necessarily be right for all projects, given the constraints under which product developers live, it is a key issue to consider when the practice that we are "reducing" to is not just for AI research but for development and product.

In sum, a number of key factors of a strongly pragmatic sort show that logics that look good on paper may have a long way to go before they can have any impact in the real world. These factors range from upward compatibility and system maintenance to implementation tradeoffs and critical system features like error-handling and explanation. They include learnability of the language and occasional escapes to circumvent limitations of the system. While individual gains in CLASSIC derived from attention to these practical concerns may each have been small, they all added up, and made a big difference to success of the system as a whole. It is quite clear that if practical concerns were ignored, the resulting system would have had at best limited utility. In fact, in general in our field, it seems that the true theoretical work is not done until the implementation runs and the users have had their say.

References

1. Aït-Kaci, H.: Type subsumption as a model of computation. In: Kerschberg, L. (ed.) Proceedings of the First International Conference on Expert Database Systems, Kiawah Island, South Carolina, October 1984, pp. 124–150 (1984)
2. Baader, F., Hollunder, B., Nebel, B., Profitlich, H.-J., Franconi, E.: An empirical analysis of optimization techniques for terminological representation systems, or, making KRIS get a move on. In: Nebel, et al. (eds.) [35], pp. 270–281
3. Baader, F., Küsters, R., Borgida, A., McGuinness, D.: Matching in description logics. Journal of Logic and Computation 9(3), 411–447 (1999)

4. Borgida, A.: Extensible knowledge representation: the case of description reasoners. Journal of Artificial Intelligence Research 10, 399–434 (1999)
5. Borgida, A., Brachman, R.J., McGuinness, D.L., Resnick, L.A.: CLASSIC: A structural data model for objects. In: Proceedings of the 1989 ACM SIGMOD International Conference on Mangement of Data, June 1989, pp. 59–67. Association for Computing Machinery, New York (1989)
6. Borgida, A., Küsters, R.: What's NOT in a name?: Initial explorations of a structural approach to intgerating large concept knowledge bases. Technical Report DCS-TR-391, Rutgers University, Dept. of Computer Science (August 1999)
7. Borgida, A., Patel-Schneider, P.F.: A semantics and complete algorithm for subsumption in the Classic description logic. Journal of Artificial Intelligence Research 1, 277–308 (1994)
8. Borgida, A., McGuinness, D.L.: Inquiring about frames. In: Aiello, L.C., Doyle, J., Shapiro, S.C. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR 1996), November 1996, pp. 340–349. Morgan Kaufmann Publishers, San Francisco (1996)
9. Brachman, R.J.: A Structural Paradigm for Representing Knowledge. PhD thesis, Harvard University, Cambridge, MA (1977); BBN Report No. 3605, Bolt Beranek and Newman, Inc., Cambridge, MA (July 1978) (revised version)
10. Brachman, R.J., Fikes, R.E., Levesque, H.J.: KRYPTON: Integrating terminology and assertion. In: Proceedings of the Third National Conference on Artificial Intelligence, Washington, DC, August 1983, pp. 31–35. American Association for Artificial Intelligence, Menlo Park (1983)
11. Brachman, R.J., Levesque, H.J.: The tractability of subsumption in frame-based description languages. In: Proceedings of the Fourth National Conference on Artificial Intelligence, Austin, Texas, August 1984, pp. 34–37. American Association for Artificial Intelligence, Menlo Park (1984)
12. Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A., Borgida, A.: Living with CLASSIC: When and how to use a KL-ONE-like language. In: Sowa, J.F. (ed.) Principles of Semantic Networks: Explorations in the representation of knowledge, pp. 401–456. Morgan Kaufmann Publishers, San Francisco (1991)
13. Brachman, R.J., Schmolze, J.G.: An overview of the KL-ONE knowledge representation system. Cognitive Science 9(2), 171–216 (1985)
14. Brachman, R.J., Selfridge, P.G., Terveen, L.G., Altman, B., Borgida, A., Halper, F., Kirk, T., Lazar, A., McGuinness, D.L., Resnick, L.A.: Knowledge representation support for data archaeology. In: First International Conference on Information and Knowledge Management, Baltimore, MD, November 1992, pp. 457–464 (1992)
15. Bresciani, P., Franconi, E., Tessaris, S.: Implementing and testing expressive description logics: a preliminary report. In: Ellis, G., Levinson, R.A., Fall, A., Dahl, V. (eds.) Knowledge Retrieval, Use and Storage for Efficiency: Proceedings of the First International KRUSE Symposium, pp. 28–39 (1995)
16. Chaudhri, V.K., Farquhar, A., Fikes, R., Karp, P.D.: Open Knowledge Base Connectivity 2.0. Technical report, Technical Report KSL-09-06, Stanford University KSL (1998)
17. Chaudhri, V.K., Farquhar, A., Fikes, R., Karp, P.D., Rice, J.: The Generic Frame Protocol 2.0. Technical report, Artificial Intelligence Center, SRI International, Menlo Park, CA (July 1997)
18. Devanbu, P., Brachman, R.J., Ballard, B., Selfridge, P.G.: LaSSIE: A knowledge-based software information system. Communications of the ACM 34(5), 35–49 (1991)

19. Donini, F.M., Lenzerini, M., Nardi, D., Nutt, W.: Tractable concept languages. In: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence. International Joint Committee on Artificial Intelligence, Sydney, Australia, August 1991, pp. 458–453 (1991)
20. Donini, F.M., Lenzerini, M., Nardi, D., Schaerf, A., Nutt, W.: Adding epistemic operators to concept languages. In: Nebel et al. (ed.) [35], pp. 342–353
21. Doyle, J., Patil, R.: Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. Artificial Intelligence 48(3), 261–297 (1991)
22. Horrocks, I.: Using an expressive description logic: FaCT or fiction? In: Cohn, A.G., Schubert, L., Shapiro, S.C. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR 1998), June 1998, pp. 636–647. Morgan Kaufmann Publishers, San Francisco (1998)
23. International Joint Committee on Artificial Intelligence. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (August 1995)
24. MacGregor, R.M.: A deductive pattern matcher. In: Proceedings of the Seventh National Conference on Artificial Intelligence, St. Paul, Minnesota, August 1988, pp. 403–408. American Association for Artificial Intelligence, Menlo Park (1988)
25. McGuinness, D.L.: Explaining Reasoning in Description Logics. PhD thesis, Department of Computer Science, Rutgers University (October 1996); also available as Rutgers Technical Report Number LCSR-TR-277
26. McGuinness, D.L.: Ontological issues for knowledge-enhanced search. In: Proceedings of Formal Ontology in Information Systems. IOS-Press, Washington (1998); also In: Frontiers in Artificial Intelligence and Applications (to appear)
27. McGuinness, D.L., Borgida, A.: Explaining subsumption in Description Logics. In: IJCAI 1995 [23], pp. 816–821
28. McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A., Isbell, C., Parker, M., Welty, C.: A description logic-based configuration for the web. SIGART Bulletin 9(2) (fall, 1998)
29. McGuinness, D.L., Resnick, L.A., Isbell, C.: Description Logic in practice: A CLASSIC application. In: IJCAI-1995 [23], pp. 2045–2046
30. McGuinness, D.L., Wright, J.R.: Conceptual modeling for configuration: A description logic-based configurator platform. Artificial Intelligence for Engineering Design, Analysis, and Manufacturing Journal - Special Issue on Configuration (1998)
31. McGuinness, D.L., Wright, J.R.: An industrial strength description logic-based configuration platform. IEEE Intelligent Systems (1998)
32. Moser, M.G.: An overview of NIKL, the new implementation of KL-ONE. Technical Report 5421, BBN Laboratories, 1983. Part of a collection entitled "Research in Knowledge Representation for Natural Language Understanding—Annual Report (September 1, 1982–August 31, 1983)
33. Mylopoulos, J., Bernstein, P., Wong, H.K.T.: A language facility for designing database-intensive applications. ACM Transactions on Database Systems 5(2), 185–207 (1980)
34. Nebel, B.: Terminological reasoning is inherently intractable. Artificial Intelligence 43(2), 235–249 (1990)
35. Nebel, B., Rich, C., Swartout, W. (eds.): Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR 1992). Morgan Kaufmann Publishers, San Francisco (1992)

36. Patel-Schneider, P.F.: Small can be beautiful in knowledge representation. In: Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, Denver, Colorado, December 1984, pp. 11–16. IEEE Computer Society, Los Alamitos (1984)
37. Patel-Schneider, P.F.: Undecidability of subsumption in NIKL. Artificial Intelligence 39(2), 263–272 (1989)
38. Patel-Schneider, P.F.: DLP system description. In: Franconi, E., De Giacomo, G., MacGregor, R.M., Nutt, W., Welty, C.A. (eds.) Proceedings of the 1998 International Workshop on Description Logics, June 1998, pp. 87–89 (1998); available electronically as a CEUR publication at
`http://SunSite.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-11/`
39. Peltason, C., von Luck, K., Nebel, B., Schmiedel, A.: The user's guide to the BACK system. KIT-Report 42, Fachbereich Informatik, Technische Universität Berlin (January 1987)
40. Schaerf, A.: Reasoning with individuals in concept languages. Data and Knowledge Engineering 13(2), 141–176 (1994)
41. Schmidt-Schauss, M.: Subsumption in KL-ONE is undecidable. In: Brachman, R.J., Levesque, H.J., Reiter, R. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the First International Conference (KR 1989), May 1989, pp. 421–431. Morgan Kaufmann Publishers, San Francisco (1989)
42. Selfridge, P.: Knowledge representation support for a software information system. In: IEEE Conference on Artificial Intellingence Applications, Miami, Florida, February 1991, pp. 134–140. The Institute of Electrical and Electronic Engineers (1991)
43. von Luck, K., Nebel, B., Peltason, C., Schmiedel, A.: BACK to consistency and incompleteness. In: Stoyan, H. (ed.) Proceedings of GWAI-1985—the 9th German Workshop on Artificial Intelligence, pp. 245–257. Springer, Heidelberg (1986)
44. Wright, J.R., Weixelbaum, E.S., Brown, K., Vesonder, G.T., Palmer, S.R., Berman, J.I., Moore, H.H.: A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T network systems. In: Proceedings of the Innovative Applications of Artificial Intelligence Conference, Washington, July 1993, pp. 183–193. American Association for Artificial Intelligence, Menlo Park (1993)