SIGART Bulletin **2**, 3 (1991), pages 108–113.

The CLASSIC Knowledge Representation System: Guiding Principles and Implementation Rationale

Peter F. Patel-Schneider
Ronald J. Brachman
Deborah L. McGuinness
Lori Alperin Resnick
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

Alex Borgida
Department of Computer Science
Rutgers University
New Brunswick, New Jersey

6 December 1991

Abstract

Our work on the CLASSIC knowledge representation system covers a broad range from theory to practice. While CLASSIC was implemented primarily to provide a simple, easy to learn and use, locally available tool for a relatively limited set of applications, it has a substantial theoretical foundation, based on a formal "terminological" logic. The logical foundation provides the semantics of a term description language, which is used to define structured concepts and make assertions about individuals in a knowledge base. These concepts and individuals are organized into a generalization hierarchy by classification and subsumption algorithms. The CLASSIC system explores the expressiveness vs. tractability tradeoff, driven by concerns of usefulness and usability in several real applications. Within this context, it embodies our views of what a knowledge representation system should be: useful, comprehensible and usable, predictable, non-subvertable, and at bottom, based on a formal logic.

The CLASSIC knowledge representation system [1, 2] has been designed and implemented at AT&T Bell Laboratories over the last several years. CLASSIC has its roots in at least a dozen years of work on theory and implementation of

"terminological" knowledge representations.[1] The main reason for building an implementation was to provide a locally available tool to support several classes of applications that were important at AT&T. Among our main concerns were leverage on these applications—that is, the system had to be *useful* in a real practical context—it had to be understandable by non-experts—that is, really *usable*—and it had to be *reliable* and *predictable*.

CLASSIC is based on a formal logic, research on which proceeds in parallel with implementation and use, and there has been much interplay between these two sides of the work. We have now had significant practical experience with the first release of the system, and are substantially into a second version. CommonLisp and C implementations of CLASSIC exist and are in use in applications. The system is fully documented [9] and the CommonLisp version is available to universities for research use.[2] CLASSIC has been used in several university courses, and is taught in an internal Bell Labs course on knowledge representation.

1 Guiding Principles

The design of the CLASSIC system, like that of all computer tools (e.g., programming languages and database management systems), is the result of compromises between often contradictory desires: maximum expressive and reasoning power, efficiency, ease of learnability and usability, etc. In developing CLASSIC, we have been guided by a number of principles, which help distinguish it from many other implemented KR&R systems.

The fundamental principle underlying the design of CLASSIC is that *knowledge representation and reasoning systems should be predictable.* CLASSIC achieves predictability by providing its users with a clear description of its behavior *independent of its implementation* (i.e., one can describe the inferences made by the system in any situation without having to execute or trace our implementation code for CLASSIC), and by having all operations return results in a reasonable amount of time (i.e., the time taken to process questions can be easily predicted from the form of the facts told to the system, and does not involve potentially unbounded search).

The predictability of inferences is supported by a second guideline: *a knowledge representation system should be based on a formal logic.* Such a logic normally provides a language for expressing assertions (as well as questions and maybe even answers) and a notion of *entailment* relating responses (and non-responses) provided by the system. Usually logics are described in at least two

[1] Other systems in CLASSIC's immediate family include KANDOR [7] and BACK [8]

[2] We have released CLASSIC to over 20 universities. In order to request a copy of the CommonLisp version, send a letter to D. McGuinness at the above address. This should be on university letterhead and it should state that CLASSIC will be used for research and/or educational purposes only.

complementary ways: model-theoretically (the denotational semantics of the sentences) and proof-theoretically (syntactic rules of inference). Together with the implementation of the reasoner, this provides three perspectives for understanding the intended behavior of the KR system, which leads to much better comprehension than with any one alone.

An ideally useful general KR system would then provide maximum expressive power and be based on some form of logic. Unfortunately, it is well known that increased expressiveness comes at the cost of resource utilization, even non-computability. While some applications of knowledge representation demand comprehensive expressive power (in exchange perhaps for incomplete or user-controlled inference), not all do. We believe that in a reasonable and important class of applications, *trading expressive power for computational tractability by providing a simple and limited language is the right choice to make.* Besides directly supporting our goal of predictability, this principle provides significant additional benefits: simpler languages are easier to learn and to use correctly, as well as to implement, maintain and port. Of course, for some kinds of applications this principle is not the right one to follow: for example, natural language understanding and general-purpose logic programming would seem to require Turing-equivalence.[3] However, it is now clear to us from several significant industrial applications that arbitrary expressive power is simply not always necessary.

A final principle that has been followed in the design of CLASSIC is that *there should be no way of subverting the knowledge representation system.* This means that user code should not destroy the correspondence between the reasoning subsystem and its formal description.

As stated earlier, in building a useful and usable system one has to make compromises. So how have the above principles fared? With two exceptions, to be discussed in a moment, all of the features of the CLASSIC language are implementations of aspects of a formal logic. Moreover, the formal computational complexity of subsumption reasoning with these features is known: the only constructor whose standard semantics would require intractable reasoning is **ONE-OF**—sets of individuals—and the reasoning actually implemented by the system is both polynomial time and describable by a precise, albeit slightly nonstandard semantics, as well as formal rules of inference.[4] We therefore have a limited and tractable logic. Moreover, the current implementation reasons in a sound and complete manner with respect to even the standard semantics of the **ONE-OF** constructor, if the elements of the set are *host-language individuals*, which cannot have contingent properties. The practical systems built with

[3] CLASSIC is also unsuitable for heuristic classification applications because the logic it is based upon is two-valued, and does not deal with uncertainty.

[4] Theoretically, it is possible that expanding concept identifier definitions might also lead to an explosion in size [5] but this has never been observed in practice either in AI or in the extensive use of the SML programming language, which has a similar problem in its type system.

CLASSIC have been able to observe this guideline.

While our goal was to build a direct implementation of our logic, it was evident that in different situations users would want to express a variety of concepts and inferences not provided by that logic. Rather than try to anticipate in advance and design in all possible desired features, we chose to maintain usability by adding to CLASSIC two simple features: (1) a limited forward-chaining rule system, whose interpretation is somewhat "procedural" (i.e., "if c_1 is added to the description of individual i then also add c_2"), and (2) concepts whose definitions include tests written as functions in the host programming language (and which are therefore opaque to such operations as concept subsumption).

These aspects of CLASSIC are therefore outside the scope of the formal logic, yet they are *limited*: the rules are of a very simple, monotonic kind—certainly not a full rule system; and even though user code is allowed to be part of concept descriptions, CLASSIC maintains the distinction between its own operations and other computations: there are clear rules for what these test functions can and cannot do, which, if followed, will result in no possibility of subverting the semantics of the rest of the concept definitions.

In some respects, it is these features that most distinguish CLASSIC from its precursor, KANDOR, which also subscribed to many of the above principles, though it had somewhat different constructors, and its inference was incomplete in several places.

2 Comments on the Implementation

As mentioned, the primary reason for implementing CLASSIC was to provide inferential support for several key applications—in simple terms, we needed a usable KR tool. In addition, although CLASSIC was designed as the implementation of a term description language, with clear *a priori* theoretical properties, there are a number of aspects of the logic underlying it that can not really be evaluated without an implemented (and used) system. The implementation itself demonstrates that some theoretically interesting portions of the terminological logic can be built to run acceptably fast in real domains. This to some extent addresses the distinction between normal case and worse case analysis, as well as the importance of constant factors—both of which are usually ignored by formal complexity analysis.

Some aspects of CLASSIC are theoretically uninteresting, and have been included so that it can be of practical use. The least interesting—but nonetheless important—of these aspects is the extensive programming interface CLASSIC presents; this interface is vital for reasonable use of the system.[5] The termino-

[5] This points out a key difference between two faces of knowledge representation: one can study and assess the "data model" and its virtues without regard to implementation, and implementation demands certain critical features not worth discussing at the data model level.

logical logic of CLASSIC also includes concepts that describe LISP objects. Although there are some theoretical problems in incorporating these *host concepts* in a terminological logic, they were included for their utility in applications, not because of any theoretical interest. Similarly, although there are some theoretically interesting aspects to the aforementioned test procedures, they were incorporated simply to allow users to extend the expressive power of CLASSIC if necessary.

The CLASSIC system has been used in three classes of applications and its basic *raison d'etre* is to support these and similar types of applications well. The first, *query by reformulation and structured knowledge browsing*, was predicted by past experience with KANDOR and ARGON. LASSIE [3] is a knowledge-based information system built to browse a large software system in an attempt to help programmers overcome the "discovery problem." The use of reasoning and query by reformulation helped minimize several identified sources of "invisibility" of software information. This work subsequently led to another effort, CODE-BASE [10], which uses CLASSIC as a semantic data model for very large databases of software information. Again, this system uses CLASSIC in a browsing fashion, but CLASSIC taxonomies also serve as *extensible schemas* for large collections of facts stored in a database. This extensibility is used to add structure to an ill-structured collection of facts about the procedures in very large computer systems. A third type of use of CLASSIC has been in *configuration* problems, primarily as an integrity checker. One system implemented in CLASSIC is currently being used on factory floors to configure transmission equipment [11]. The CLASSIC knowledge base encodes all the constraints between parts and is used to generate consistent parts lists.

The use of CLASSIC in applications has shown that it is possible to design a principled knowledge representation and reasoning system that is also useful in practice. It has also reinforced our beliefs in our principles: The formal basis of CLASSIC has served as an independent explanation of its behavior in applications, and has helped application developers to understand the system. The predictable nature of CLASSIC has allowed it to be treated as a competent piece of a much larger software system. The limited expressive power has not been an impediment to representing several interesting domains in CLASSIC. (Some domains have required the use of user-defined functions in concept definitions, but none have required extensive use of this feature.) There has been no need to subvert the meaning of CLASSIC operations through user-written functions that modify the basic behavior of the system.

CLASSIC's availability for use has also produced an unexpected benefit. We originally pictured it as being used by asserting (and retracting) information and making queries about the current state of the knowledge base. In this manner of use contradictions would seldom occur, if ever. However, several of the applications of CLASSIC within AT&T have been in configuring pieces of equipment. In these contexts information is added to a knowledge base and is

checked for consistency, i.e., the most important thing to know is whether an individual satisfies the requirements of a concept to which it is known to belong. Often this will not be the case and a contradiction is detected (and the system removes the information that triggered the contradiction). We had not originally thought of this manner of use of CLASSIC. (Unknown to us, Owsnicki-Klewe had proposed just such a use for terminologically-based knowledge representation systems [6].)

This and other additional viewpoints on representation that we obtained because others used CLASSIC have turned out to be a fruitful aspect of implementation. Aside from the unexpected benefit just mentioned, we modified the design of CLASSIC in a number of ways to accommodate users, while adhering to the same basic philosophy. We are currently in the process of building a new version of CLASSIC, with increased expressive power. The directions in which expressive power has been augmented have been largely driven by the needs of our users.[6] We plan to continue this interaction with users, but without compromising our representational guidelines in meeting their needs.

3 System Overview

CLASSIC allows the definition of roles and concepts, the association of rules with concepts, the creation of individuals, the addition of information to individuals, the retraction of information from individuals, and the retraction of rules from concepts.

In the first version of CLASSIC, roles are atomic, i.e., there is no role taxonomy. Roles are, however, divided into multi-valued roles and uni-valued roles (attributes). This distinction has important computational consequences: one construct (**SAME-AS**) is limited to paths of attributes in order to preserve tractability.

A concept in CLASSIC is defined by associating its name with it a description, composed from a simple, compositional concept description language. Concept definitions must be non-circular (i.e., the concept being defined cannot have been referenced in any existing concept description, nor can it reference itself). Also, concepts cannot be modified after they are created.

The concept description language forms the core of CLASSIC's representational repertoire. The syntax of concept descriptions is defined using the following grammar:

<concept-description> ::=
  `THING | CLASSIC-THING | HOST-THING |`
  (built-in host concepts) |
  <concept-name> |

[6] In fact, for most of its life, CLASSIC has been developed jointly with a Development group, which includes Gregg Vesonder, Elia Weixelbaum, and Jon Wright.

(**AND** <concept-expr>$^+$) |
(**ALL** <role-expr> <concept-expr>) |
(**AT-LEAST** <positive-integer> <role-expr>) |
(**AT-MOST** <non-negative-integer> <role-expr>) |
(**SAME-AS** <attribute-path> <attribute-path>) |
(**TEST-C** <fn> <argument>*) |
(**TEST-H** <fn> <argument>*) |
(**ONE-OF** <individual-name>*) |
(**PRIMITIVE** <concept-expr> <index>) |
(**DISJOINT-PRIMITIVE** <concept-expr>
<group-index> <index>) |
(**FILLS** <role-expr> <individual-name>$^+$)

<concept-name> ::= <symbol>
<individual-name> ::= <symbol> | <cl-host-expr>
<role-expr> ::= <mrole-expr> | <attribute-expr>
<mrole-expr> ::= <symbol>
<attribute-path> ::= (<attribute-expr>$^+$)
<attribute-expr> ::= <symbol>
<cl-host-expr> ::= <string> | <number> |
'<CommonLISP-expr> |
(quote <CommonLISP-expr>)
<index> ::= <number> | <symbol>
<group-index> ::= <number> | <symbol>
<fn> ::= a three-valued logical function in the
host language (Common LISP)
<argument> ::= an expression passed to the test function

Most of this syntax is similar to that used in related systems (e.g., BACK and LOOM [4]). Some important less familiar features are briefly described here.

The **SAME-AS** constructor requires that the two composed attribute (univalued role) paths have the same filler. The **TEST-C** and **TEST-H** constructors allow users to write functions that serve as tests for membership in concepts ("**C**" being for CLASSIC individuals and "**H**" for host individuals). The **PRIMITIVE** constructor allows the creation of concepts with incomplete definitions. (The <index> portion of the constructor allows identification of different primitive concept descriptions without having to name them.) The **DISJOINT-PRIMITIVE** constructor allows the creation of primitive concepts that are also disjoint. (Disjoint primitives that only differ on their <index> argument are disjoint.)

CLASSIC has a simple forward-chaining rule mechanism. Descriptions can be attached to concepts as rule consequents and, when an individual is known to be an instance of the antecedent concept, the information in the consequent is added to the individual.

An individual is created in CLASSIC by providing a concept description that

describes it. Information, in the form of a concept description, can be added to an individual after it is created. It is also possible to state that the currently known fillers of a role for an individual are the only fillers (we call this "closing" a role). Information can also be removed from an individual and all unsupported facts are removed.

CLASSIC allows quite a number of queries about concepts, roles, and individuals. The most general queries ask whether one concept description is more general than (subsumes) another or whether an individual is described by (is an instance of) a concept description. The former query only uses information from concept definitions—no properties currently known about individuals are taken into account.

CLASSIC determines subsumption relationships between concepts when they are defined, maintaining a cached subsumption taxonomy of all named concepts. CLASSIC also determines which named concepts describe each individual, and runs each rule that applies to the individual. This means that CLASSIC detects incoherent concepts (concepts whose concept description can never have any instances) when they are defined and detects inconsistent individuals when they are created or when the information that makes them inconsistent is first added. In CLASSIC queries can never cause new incoherencies or inconsistencies.

The operations of CLASSIC have descriptions of varying formality. There is a standard model-theoretic semantics for the above concept description language which induces a standard semantics for all the operations of CLASSIC except removal of information. Removal of information is not part of this semantics because it depends on how the information was provided, and not just on the model-theoretic consequences of the information.

However, the operations of CLASSIC are incomplete with respect to this semantics. Subsumption does not take into account the semantics of the host functions and also does not take into account properties of individuals implied by concept definitions (although note the comment above about its completeness in practice with respect to **ONE-OF**). A variant semantics has been developed for which CLASSIC's subsumption algorithm is complete.

Determining whether an individual is an instance of a concept description sometimes uses the incomplete subsumption algorithm and also does not take into account all properties of individuals that have not yet been created. A variant semantics has not yet been developed to describe this processing.

The above description shows that there are a number of aspects of the CLASSIC system that are not defined by CLASSIC's theoretical underpinnings. Moreover, even the theoretical underpinnings of CLASSIC were influenced by the goal of producing an efficient and usable implemented system.

4 Annotated Example

The following example demonstrates some typical uses of CLASSIC. It should also help to describe the syntax of CLASSIC.

Because of space considerations, the following is not a complete transcript and may not precisely conform to the actual workings of CLASSIC. It is, however, taken from an real transcript, and is closely related to the examples used in several papers on CLASSIC. The example uses the domain of wines and meals (for a longer example in the same domain see [2]).

Most of the example is devoted to defining the ontology of the domain. First, the roles are defined.

```
(cl-define-roles course grape)
(cl-define-attributes
    color body flavor sugar maker drink food)
```

Recall that attributes are nothing more than single-valued roles.

Concepts that will be used as value restrictions for roles are defined next. CLASSIC does not allow ranges for roles to be stated, except by means of value restrictions in concept definitions.

```
(cl-define-concept 'wine-color
    '(one-of white rose red))
(cl-define-concept 'wine-body
    '(one-of light medium full))
(cl-define-concept 'wine-flavor
    '(one-of delicate moderate strong))
(cl-define-concept 'wine-sugar
    '(one-of sweet off-dry dry))
```

Now the top levels of the domain ontology are created. There are several sets of disjoint primitives in the top levels; a disjoint primitive is only disjoint from other disjoint primitives under the same parent concept and with the same group index.

```
(cl-define-concept 'consumable-thing
    '(disj-prim classic-thing 1 1))
(cl-define-concept 'winery
    '(disj-prim classic-thing 1 2))

(cl-define-concept 'edible-thing
    '(disj-prim consumable-thing 1 1))
(cl-define-concept 'potable-liquid
    '(disj-prim consumable-thing 1 2))
(cl-define-concept 'meal-course
    '(and (disj-prim consumable-thing 1 3)
```

```
          (all food edible-thing)
          (all drink potable-liquid)))
```

`meal-course` is the first concept that actually has restrictions.

Next some types of food and particular foods are defined:

```
(cl-define-concept 'fruit
    '(disj-prim edible-thing 1 1))
(cl-define-concept 'dessert
    '(disj-prim edible-thing 1 2))
(cl-define-concept 'seafood
    '(disj-prim edible-thing 1 3))
(cl-define-concept 'shellfish
    '(disj-prim seafood 1 1))
(cl-define-concept 'fish
    '(disj-prim seafood 1 2))
(cl-define-concept 'oyster-shellfish
    '(disj-prim shellfish 1 1))
(cl-define-concept 'non-oyster-shellfish
    '(disj-prim shellfish 1 2))

(cl-create-ind 'oysters 'oyster-shellfish)
(cl-create-ind 'crab 'non-oyster-shellfish)
(cl-create-ind 'cake 'dessert)
```

Now the grapes are given similar treatment.

```
(cl-define-concept 'grape
    '(prim fruit 1))
(cl-define-concept 'wine-grape
    '(prim grape 1))

(cl-create-inds '(chardonnay
    sauvignon-blanc semillon
    cabernet-sauvignon) 'wine-grape)
```

Finally we get into the wines. First, general categories of wine are defined.

```
(cl-define-concept 'wine
    '(prim (and potable-liquid
           (all color wine-color)
           (all body wine-body)
           (all flavor wine-flavor)
           (all sugar wine-sugar)
           (at-least 1 grape)
           (all grape wine-grape)
```

```
            (all maker winery))
      1))

(cl-define-concept 'red-wine
    '(and wine (fills color red)))
(cl-define-concept 'white-wine
    '(and wine (fills color white)))
```

Second, some basic types of wine are given. The third and subsequent arguments to `cl-define-concept` are rules associated with the concept. Because the types of wines are primitive concepts, there is not all that much difference between a rule and an extra piece of the concept description.

```
(cl-define-concept 'chardonnay
    '(prim white-wine chardonnay)
    '(all grape (one-of chardonnay))
    '(all body (one-of full medium))
    '(all flavor (one-of strong moderate)))
(cl-define-concept
    'semillon-or-sauvignon-blanc
    '(prim wine semillon-or-sauvignon-blanc)
    '(fills color white)
    '(all grape
          (one-of semillon sauvignon-blanc))
    '(all body (one-of medium full)))
(cl-define-concept 'cabernet-sauvignon
    '(prim red-wine cabernet-sauvignon)
    '(all grape (one-of cabernet-sauvignon))
    '(all flavor (one-of moderate strong))
    '(all body (one-of medium full))
    '(fills sugar dry))
```

Third, actual wines are defined. Many wines have been omitted to save space.

```
(cl-create-ind 'forman-chardonnay
    '(and chardonnay
          (fills body full)
          (fills flavor moderate)
          (fills sugar dry)
          (fills maker forman)))
(cl-create-ind
    'kalin-cellars-sauvignon-blanc
    '(and sauvignon-blanc
          (fills body full)
```

```
          (fills flavor strong)
          (fills sugar dry)
          (fills maker kalin-cellars)))
(cl-create-ind 'kalin-cellars-semillon
    '(and semillon
          (fills body full)
          (fills flavor strong)
          (fills sugar dry)
          (fills maker kalin-cellars)))
(cl-create-ind
    'schloss-volrad-trochenbierenauslese-riesling
    '(and riesling
          (fills body full)
          (fills flavor moderate)
          (fills color white)
          (fills sugar sweet)
          (fills maker schloss-volrad)))
(cl-create-ind
    'schloss-rothermel-trochenbierenauslese-riesling
    '(and riesling
          (fills body full)
          (fills flavor strong)
          (fills color white)
          (fills sugar sweet)
          (fills maker schloss-rothermel)))
```

Now do the same for courses in meals.

```
(cl-define-concept 'dessert-course
    '(and meal-course (all food dessert))
    '(all drink (and (fills body full)
                     (fills flavor strong)
                     (fills sugar sweet))))
(cl-define-concept 'seafood-course
    '(and meal-course (all food seafood))
    '(all drink white-wine))
(cl-define-concept 'shellfish-course
    '(and meal-course (all food shellfish))
    '(all drink
          (and (fills body full)
           (all flavor
                (one-of moderate strong)))))
(cl-define-concept 'oyster-shellfish-course
    '(and meal-course
          (all food oyster-shellfish))
```

```
    '(all drink (fills sugar sweet)))
(cl-define-concept
    'non-oyster-shellfish-course
    '(and meal-course
          (all food non-oyster-shellfish))
    '(all drink (fills sugar dry)))
```

The rules attached to these concepts perform most of the actual work in this domain. For example, if a meal-course is recognized as a dessert-course, the drink of that course is restricted to be a full-bodied, strong-flavored, sweet wine.

Now the domain of wines and meal courses has been defined. A particular use of this application would proceed by creating meal courses and determining which wines are compatible with the course. (In the larger application meal courses are grouped into meals or picnic baskets and more compatibilities can be stated and checked.)

A simple meal course would be:

```
(cl-create-ind 'course-1
    '(and meal-course (fills food crab)))
```

This individual is automatically placed under the appropriate concepts in the concept taxonomy, including, in this case, `seafood-course`, `shellfish-course`, and `non-oyster-shellfish-course`, and any applicable rules are run, resulting in information being determined about the `drink` role of the individual.

The `drink` role of this individual can be queried in several ways. First, the properties of any filler of the role can be determined—in this case any filler must be a white, full-bodied, moderate or strong-flavored, dry wine. Second, the currently known individual wines that can be fillers can be determined—in this case resulting in Forman chardonnay, Kalin semillon, and Kalin sauvignon blanc.

A more complicated example involves creating two meal courses with the same drink. This can be done by explicitly creating a dummy individual to be the drink as follows:

```
(cl-create-ind 'course-24
    '(and meal-course (fills food oysters)
                      (fills drink wine-1)))
```

At this point the drink must be white, full-bodied, moderate- or strong-flavored, and sweet. There are two known wines that match this description, the two rieslings.

When the second course is added with the same drink

```
(cl-create-ind 'course-25
    '(and meal-course (fills food cake)
                      (fills drink wine-1)))
```

the drink is further restricted to be strong-flavored. The only wine that matches this description is Schloss Rothermel trochenbierenauslese riesling.

Suppose we didn't know exactly what food we were going to serve at a course but we did know something more general like the class of the food. Say for example, instead of oysters, we know that we are going to have a seafood meal. We would retract `oysters`[7] and add seafood meal.

The oysters are retracted by

```
(cl-ind-remove-filler @course-24
    @food @oysters)
```

This change results in the drink not having a color restriction, as this had come from the fact that oysters were being served. (This shows a major difference between rules in CLASSIC and rules in many other forward chaining rule-based systems.) The seafood course information is added by

```
(cl-ind-add @course-24 '(all food seafood))
```

again restricting the color of the drink to white.

If we now tried to specialize the seafood meal to crab

```
(cl-ind-add-filler @course-24 @food @crab)
```

there would be a contradiction since non-oyster shellfish courses, including crab courses, require dry wines and we already know that the wine here is restricted to be sweet (because it is also the drink for a cake course). CLASSIC will produce an error and refuse to add this information.

If crab were really desired as the food for `course-24`, then either the two courses would have to have different drinks or the food for `course-25` would have to be changed. This type of contradiction resolution could be done interactively but is not currently done automatically by CLASSIC which instead simply removes the most recent information added.

This process of adding information to an individual or group of individuals and performing contradiction resolution when necessary is precisely the method that is used in the configuration applications developed at AT&T.

5 Current Research

Our experiences with CLASSIC's logic, its implementation and use have spurred us to continue to refine our view of such systems. In particular, there is ongoing work on numerous fronts related to CLASSIC:

[7] Note that retracting `oysters` does not leave the system knowing anything like the class of the food for the course. The only information that was given to the system—`oysters`—was retracted so all inferred information like seafood meal is retracted along with it.

- a second version of the language, with a somewhat richer set of constructors, though still abiding by the original design principles;
- an explanation facility that can help users understand the reasoning that produced certain answers;
- a graphical user interface;
- an interactive query language for evaluating complex queries;
- a facility for creating and loading individuals from data already available in relational databases, and eventually a full persistence mechanism for CLASSIC; and
- PROTO-TL – an embryonic version of CLASSIC that can be *extended* to include new, possibly domain-dependent term constructors (e.g., related to time, etc.) by filling in template functions describing the necessary inferences.

References

[1] Alex Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Mangement of Data*, pages 59–67. Association for Computing Machinery, June 1989.

[2] Ronald J. Brachman, Deborah L. McGuinness, Peter F. Patel-Schneider, Lori Alperin Resnick, and Alex Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In John F. Sowa, editor, *Principles of Semantic Networks: Explorations in the representation of knowledge*, pages 401–456. Morgan Kaufmann Publishers, San Francisco, California, 1991.

[3] Premkumar Devanbu, Ronald J. Brachman, and Peter G. Selfridge. LaSSIE—a classification-based software information system. In *Proceedings of the International Conference on Software Engineering*, Nice, France, 1990. IEEE Computer Society.

[4] Robert M. MacGregor and Raymond Bates. The Loom knowledge representation language. Technical Report ISI/RS-87-188, Information Sciences Institute, University of Southern California, May 1987.

[5] Bernhard Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2):235–249, May 1990.

[6] Bernd Owsnicki-Klewe. Configuration as a consistency maintenance task. In W. Hoeppner, editor, *Proceedings of GWAI-88—the 12th German Workshop on Artificial Intelligence*, pages 77–87. Springer Verlag, September 1988.

[7] Peter F. Patel-Schneider. Small can be beautiful in knowledge representation. In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, pages 11–16, Denver, Colorado, December 1984. IEEE Computer Society.

[8] Christof Peltason, Albrecht Schmiedel, Carsten Kindermann, and Joachim Quantz. The BACK system revisited. KIT-Report 75, Fachbereich Informatik, Technische Universität Berlin, September 1989.

[9] Lori Alperin Resnick, Alex Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Peter F. Patel-Schneider. CLASSIC description and reference manual for the COMMON LISP implementation. AI Principles Research Department, AT&T Bell Laboratories, December 1992.

[10] Peter Selfridge. Knowledge representation support for a software information system. In *IEEE Conference on Artificial Intellingence Applications*, pages 134–140, Miami, Florida, February 1991. The Institute of Electrical and Electronic Engineers.

[11] Jon R. Wright, Elia S. Weixelbaum, Karen Brown, Gregg T. Vesonder, Stephen R. Palmer, Jay I. Berman, and Harry H. Moore. 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*, pages 183–193, Washington, D. C., July 1993. American Association for Artificial Intelligence.