Detecting Knowledge Base Inconsistencies Using Automated Generation of Text and Examples

Vibhu O. Mittal Johanna D. Moore
Learning Research & Development Center and
Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15260, USA.
e-mail: {mittal,jmoore}@cs.pitt.edu

Abstract

Verifying the fidelity of domain representation in large knowledge bases (KBs) is a difficult problem: domain experts are typically not experts in knowledge representation languages, and as knowledge bases grow more complex, visual inspection of the various terms and their abstract definitions, their interrelationships and the limiting, boundary cases becomes much harder. This paper presents an approach to help verify and refine abstract term definitions in knowledge bases. It assumes that it is easier for a domain expert to determine the correctness of individual concrete examples than it is to verify and correct all the ramifications of an abstract, intensional specification. To this end, our approach presents the user with an interface in which abstract terms in the KB are described using examples and natural language generated from the underlying domain representation. Problems in the KB are therefore manifested as problems in the generated description. The user can then highlight specific examples or parts of the explanation that seem problematic. The system reasons about the underlying domain model by using the discourse plan generated for the description. This paper briefly describes the working of the system and illustrates three possible types of problem manifestations using an example of a specification of floating-point numbers in Lisp.

Introduction

Knowledge base construction is often an iterative process of debugging and refinement. As knowledge bases (KBs) increase in size, the problems of detecting incorrect, inconsistent or incomplete specifications become increasingly difficult, especially for domain experts who may be unfamiliar with the knowledge representation language and its intricacies. To alleviate this problem, a number of previous efforts have considered approaches that would allow domain experts to inspect formal specifications using natural language, e.g., (Gil 1994; Swartout 1983). However, studies show that people can usually understand and verify specific examples more easily and quickly than abstract, textual descriptions, e.g., (Reder, Charney, & Morgan 1986; Pirolli 1991). Other approaches have considered the use of examples alone to aid in debugging, e.g., (Shapiro 1983; Mitchell, Utgoff, & Banerji 1983).

Our work integrates and extends these approaches in an interface that allows users to inspect and debug KBs by identifying problems in automatically generated examples and accompanying natural language descriptions. Using information about the specific examples flagged by the user as being problematic, information about the type of the problem (also specified by the user), and the discourse plan underlying the automatically generated presentation, the system attempts to localize the problem in the KB specification. In cases where the system cannot uniquely identify the problem with the knowledge base, it generates additional descriptions for the expert to verify. This work integrates previous research in three areas: (1) knowledge acquisition and refinement, e.g., (Gil 1994; Musen *et al.* 1988), (2) natural language generation and reasoning about discourse plans, e.g., (Moore & Paris 1993), and (3) automatic example generation, e.g., (Ashley & Aleven 1992; Mittal & Paris 1994). In our analysis, problems in the KB specification of a concept manifest themselves as a combination of one or more of the following three types of errors in system generated explanations: (1) incorrect examples, (2) incorrect explanations accompanying the examples, or (3) sequencing problems in the examples.

Examples and the accompanying textual descriptions are generated by a hierarchical discourse planner, which produces discourse plans recording the goals achieved by and the rhetorical relationships among plan components. When the user indicates that an example is incorrect (by highlighting the example), the system uses the discourse plan to generate and reason about hypotheses regarding possible errors in the KB specifications that could have led to the errors in the description generated. Our system differs from previous work on example based debugging, e.g., (Shapiro 1983; Mitchell, Utgoff, & Banerji 1983), because it uses knowledge about the discourse plan that generated the examples and accompanying text, as well as domain knowledge about near-misses, in order to localize possible problems in the KB.

To illustrate the application and utility of our approach in detecting and debugging KB problems, this paper discusses three types of errors in descriptions that can indicate problems in the underlying KB specification–wrong examples, wrong explanations and incorrect example sequencing–and how they can help in finding the problem. In order to illustrate the general problem, rather than focus on system- and representation-specific mechanisms, all of the scenarios discussed in this paper use a Backus-Naur Form (BNF) representation of the domain. BNF is a generic, domain- and task-neutral specification formalism that is capable of representing a wide variety of domains and tasks ranging from mechanical device design (Mohd-Hashim, Juster, & de Pennington 1994) to protein-structure mapping (George, Mewes,

& Kihara 1987) and interface requirements (Reisner 1981). To further simplify the discussion, we use the same BNF fragment to illustrate the three types of errors that can occur in the automatically generated presentation. For this purpose, we use the specification of floating point numbers in Lisp, but the method discussed in this paper is specific neither to Lisp, nor in fact to BNF. We have chosen this example here because (1) floating point numbers need no introduction, (2) the abstract specification of floating point numbers is sufficiently complex so as to illustrate the utility of examples, and (3) translating BNF to other KR languages has been described previously (Mittal & Paris 1994).

Generating Text and Examples

The system uses a text planner to generate coherent natural language descriptions. Given a communicative goal (such as `(DESCRIBE (CONCEPT LIST))`),[1] the system finds operators capable of achieving this goal. Operators typically post further subgoals to be satisfied, and planning continues until all goals have been refined to primitive speech acts – i.e., those directly realizable in English. The result of the planning process is a discourse plan in which the nodes represent goals at various levels of abstraction with the root being the initial goal, and the leaves representing primitive realization statements. This discourse plan is then passed to a grammar interface, which converts it into a form suitable for input to a natural language generation system, such as FUF (Elhadad & Robin 1992), to produce the surface form. The system uses a subsumption classifier, such as the one in KL-ONE based knowledge representation systems to generate the examples (Mittal & Paris 1994). A complete description of the generation system is beyond the scope of this paper – see Moore and Paris (1993) for a more detailed description of plan based natural language generation.

In order to generate and present examples that are effective in localizing problems the system categorizes each feature of the concept into one of two classes:

- **critical:** features are *required* for the example to be an instance of the concept being illustrated. For instance, by definition, a Lisp list must contain both a left- and a right-parenthesis (with the exception of `NIL`).
- **variable:** features can vary without causing the modified examples to no longer be subsumed by the definition of the concept being illustrated. For instance, the number, type and order of elements in a list in Lisp.

Given the variable and critical features of a concept, the system can use this information to plan the presentation of effective example sequences: minimally different positive-negative pairs for critical features, and groups of varying positive examples for the variable features. Determining the critical and variable features of a concept can be accomplished by using a term classifier as described in Mittal and Paris (1994), such as the ones available in the KL-ONE family of KR languages (Woods & Schmolze 1992).

To find critical and variable features of concepts defined using BNF, a straightforward way is to map the BNF definitions to KL-ONE type definitions and query the classifier. For instance, the BNF specifications[2] for floating point numbers in Lisp is given in Fig. 1. Our system maps these BNF grammar specifications into concept descriptions in the language Loom (MacGregor 1994);[3] This mapping is straightforward: non-terminal symbols in the grammar are mapped to concepts, and terminal symbols are mapped to instances. The ordering of the symbols in a production is specified by linking the respective concepts and instances using a pre-defined binary relation that the system understands as specifying the order in the BNF.

floating-point-number ::=
  [sign] {digit} decimal-point {digit}+ [exponent]* (1a)
  | *[sign] {digit}+ [decimal-point {digit}*] exponent* (1b)
sign ::= +|- (2)
decimal-point ::= . (3)
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (4)
exponent ::= exponent-marker [sign] {digit}+ (5)
exponent-marker ::= e | s | f | d | l | E | S | F | D | L (6)

Figure 1: Grammar fragment from (Steele, 1984: p.17)

In addition to presenting critical and variable features effectively–by either pairing contrasting examples or grouping similar examples–the sequence in which the examples are presented can also be important in focusing the reader's attention. For instance, presenting simpler features before more complex ones is an effective strategy, e.g., (Carnine & Becker 1982). The presentation sequence is important because users often try and understand an example in terms of others that they have seen before. In this case, the system generates examples in order of increasing complexity. In the case of BNF grammars, the measure of complexity is based on a combination of the number of productions required to generate an expression, and the complexity of each term in the expression.

The next section describes how this framework can be used to help domain experts debug and refine KBs by examining descriptions generated by the system.

Using Examples in KA

To illustrate how examples can help in detecting gaps in the KB, consider the grammar fragment shown in Fig. 1 (floating point numbers in Lisp). Even though this set of productions is one of the simpler ones in the grammar, it is easy to overlook some of the implications of the bracketing and the *kleene-* and *transitive-closures* in the productions. The rules are complex enough that a text-only paraphrase of the rules themselves may not be enough to spot a mistake in the representation. However, an example generated from *only* the faulty aspect can often stand out as a grossly wrong instance of the definition and can thus focus attention on specific aspects of

[1] The syntactic forms have been simplified for the sake of clarity.

[2] Brackets indicate optional components; braces are used for grouping things or indicating kleene (+) or transitive (*) closures.

[3] Loom is a knowledge representation language that provides classification capabilities similar to other KL-ONE languages. However, our technique is not specific to Loom.

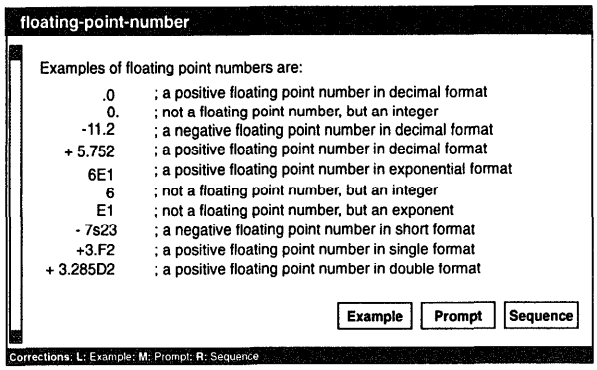

Figure 2: Description for FLOATING-POINT-NUMBER.

abstract rules in a very effective fashion (Pirolli 1991).

To generate and present the examples, the system must first determine the critical and variable features of the concept FLOATING-POINT-NUMBER. In this case, the critical features are: (i) the presence of a decimal point accompanied by one or more digits on the right hand side of the decimal point, or (ii) a number accompanied by an exponent. The variable features are: (i) the presence or absence of the sign, (ii) the value of the sign, (iii) the number of digits in the numbers, and (iv) the values of the numbers. The system can now utilize this critical/variable categorization to generate sets of examples to effectively convey each of these attributes (critical attributes by pairs of almost identical positive-negative examples; variable attributes by groups of varying positive examples). The presentation order of the examples is determined by the relative complexity of each example. A typical output generated by the system is shown in Fig. 2 (fragments of the discourse plan underlying the presentation of the critical features are shown in Fig. 3).

Now suppose that the specification of the concept `floating-point-number` is incorrect. The problems in the specification can manifest themselves in the resulting explanation that is generated in one of three ways: the examples generated by the system are incorrect, the explanations accompanying the examples are incorrect, or the examples are ordered in an inconsistent manner. (These can be marked by the domain expert as such by selecting the appropriate examples/prompts and using the 'buttons' at the bottom of the screen.) In each of these cases, the system reasons about the underlying discourse plan used to generate the explanation in order to localize the potential cause of the problem.

Case 1. A wrong example is generated: There are two possible ways in which problems in the KB manifest themselves as incorrect examples in the resulting explanation:

Case 1.1. A simple wrong example: If the faulty example differs from its adjacent (correct) examples in only a single feature, the system can use this information in conjunction with the discourse plan to debug the KB specification. Consider, again, the specification of floating-point numbers in Lisp shown in Fig. 1. The correct and one possible mistaken specification for rule ($1a$) are shown below:

floating-point-number ::=

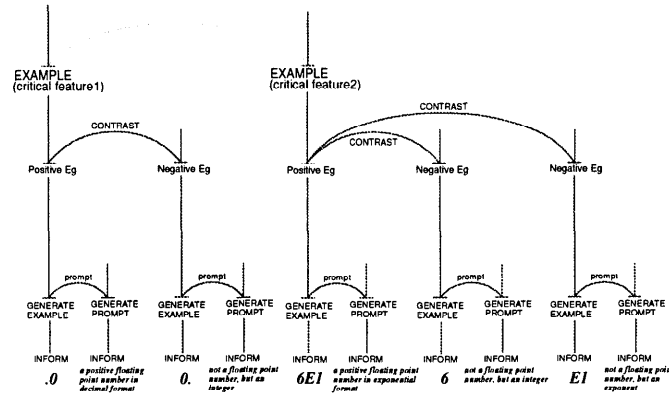

Figure 3: Fragments of the discourse plan for the two critical features.

[*sign*] {*digit*}* *decimal-point* {*digit*}+ [*exponent*] √

floating-point-number ::=

[*sign*] {*digit*}* *decimal-point* {*digit*}* [*exponent*] ×

The resulting output generated by the system for the incorrect case is shown in Fig. 4. The first and the third examples presented in the explanation are incorrect. It is clearly easier to spot the mistake in the individual examples than in the abstract specification.

Using our interface, the user can highlight these two items and indicate them as being incorrect examples of a floating point number. Based on this information, the system reasons as follows. First, it uses the discourse plan to determine which other examples in the presentation are most closely related to the items that were marked incorrect. The discourse plan indicates not only which examples are related, but *how* they are related, e.g., whether they are contrastive examples for a critical feature, similar examples for a variable feature, etc. In this case, the system determines that the first example was generated to illustrate the following variable features: the sign of the number, the number of digits on the left of the decimal point, the number of digits on the right of the decimal point, and the exponent. The second example was intended to highlight the variable nature of the digits *on the left* of the decimal point, and since that example was not marked wrong, the variable nature of the digits on the left of the decimal point is correct. The third example was supposed to illustrate the variable nature of the digits *on the right* of the decimal point, and that example was marked wrong. Since the other examples were not marked wrong, the system can, on the basis of the two wrong examples and the other correct examples, suggest a revision to the incorrect version of rule ($1a$). This revision regarding the optionality of digits on the right of the decimal point results in the transitive closure being modified to a kleene closure as follows:

floating-point-number ::=

[*sign*] {*digit*}* *decimal-point* {*digit*}+ [*exponent*]

Case 1.2. A complex wrong example: in some cases, a component term used in the example (with its own critical and variable features) can be incorrect, making the larger example wrong. When an example containing such complex

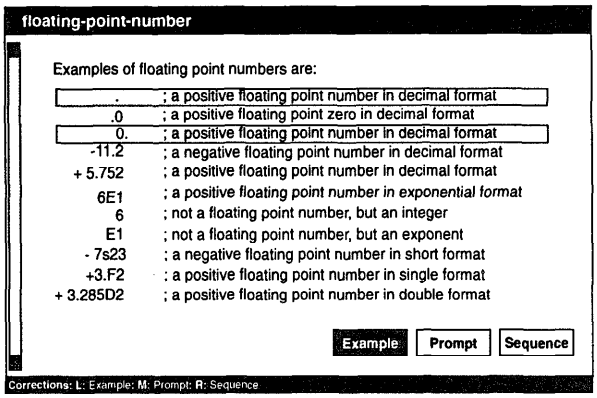

Figure 4: A simple case of incorrect examples.

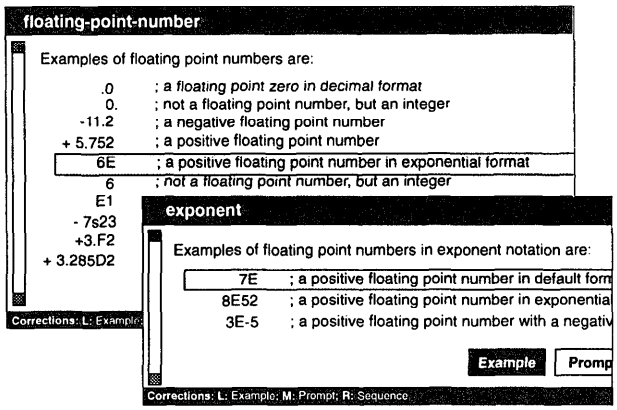

Figure 5: A complex incorrect example can result in the generation of further examples.

component terms is marked incorrect, the system can generate additional, simpler examples about the suspect component in order to localize the KB problem. Consider, for instance, the case in Fig. 5. The fifth example in the sequence, which is also the first example where the exponent notation is used, is marked as incorrect by the user. The discourse plan indicates that the example in question was generated to illustrate the use of the exponent notation in rule (1*b*). The system examines the portion of the discourse plan regarding examples generated from rule (1*b*). Since one of the differences between the wrong example and its immediate neighbor is the exponent (the *[decimal-point* {*digit*}**]* portion of the rule was not used in either of the two), the system can infer that the problem is in the specification of the exponent.

There are two other examples in the same explanation that also have exponents in them (the last two examples). These, however, use a different exponent markers ("F" and "D"). Thus, it is only possible to infer that *either the wrong marker was used, i.e., "E" is not allowed, or some other piece of information is missing.* To verify the first possibility, that "E" is an invalid exponent marker, the system generates another set of examples for floating point numbers that use the exponent marker "E" (shown in the lower half of Fig. 5). In this case, the first example of an exponent is wrong. The system can now use the discourse structure used in generating the examples for the exponent to identify the problem. In this case, the difference between the first two examples of the exponent is that the second example has a positive number following the exponent marker whereas the first example does not. Thus, one possibility is that a positive number is necessary in these cases. The third example, which has a negative number after the exponent marker, allows the system to generalize the previous hypothesis (of needing a *positive* number following the exponent) to the hypothesis that any number, positive or negative, is needed. Since the production specified that the sign is optional, the only part of the production that could be wrong is about the optionality of the number. Thus, the system can suggest that the specification of the exponent be modified to make *both* the number and the exponent marker be required in all cases:

exponent ::= exponent-marker [*sign*] {*digit*}* ×

exponent ::= exponent-marker [*sign*] {*digit*}$^+$ √

Case 2. A wrong prompt: Mistakes in the domain model can also result in the generation of incorrect textual prompts. Prompts can indicate errors in at least two cases: (*i*) the system presents a valid, positive example as being a negative, invalid example (or vice-versa), and (*ii*) the system presents a valid example (either positive or negative), but the accompanying prompt (or explanation) is either irrelevant or inconsistent with the point being illustrated.

The first possibility can be handled in the same way as in Case 1 above. However, the second possibility, where an invalid prompt is generated for a correct example, is often due to missing information, and must also be dealt with. For instance, consider the case where the system generates an example of a floating point number such as the one shown in Fig. 6. If the specification of the production rule for the exponent, (rule 5) is faulty as given below:

exponent ::= [*exponent-marker*] [*sign*] {*digit*}$^+$ ×

the system would generate the example using the second production rule for floating point numbers – the part "`5.7`" from rule (1*b*), and the digits "`5`" and "`2`" from the faulty rule given above for the exponent. Also based on the faulty rule, the system would assume that the `exponent-marker` and the `sign` were optional and therefore not to be included initially. The resulting example generated is a valid floating point number "`5.752`", *but the accompanying textual prompt indicates that a mistake was made* in the specification. Selecting the prompt causes the system to generate additional examples for the same discourse goal that caused the generation of the example with the faulty prompt in the first place. Exercising the different options of the production rule for the exponent, the system can infer that `exponent-marker` is not a variable feature, but a critical one (i.e., its presence is mandatory in the case of an exponent), and thus can propose the corrected rule:

exponent ::= exponent-marker [*sign*] {*digit*}$^+$ √

Case 3. A wrong presentation sequence: Finally, a third possible manifestation of KB problems can be seen in strange or surprising placement of examples (for instance, a simple example appearing after a number of complex examples of

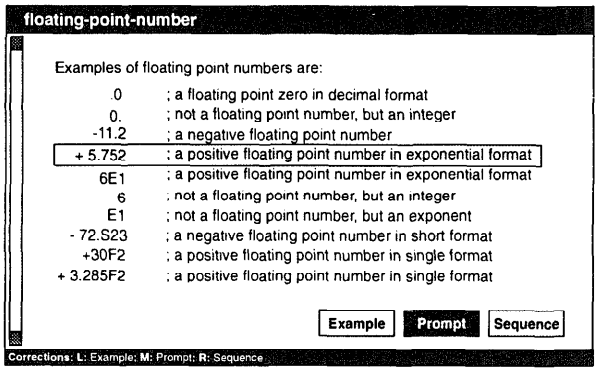

Figure 6: Errors in prompts can indicate KB problems.

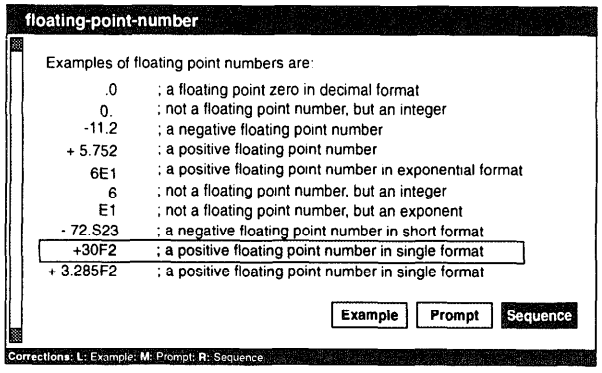

Figure 7: Bad sequencing can also indicate KB problems.

the same concept have been presented). In such cases, even though all the examples presented may be valid, the complexity assignment to each example is computed incorrectly because of the problems in the KB specifications. For instance, consider what happens if the bracketing of the transitive-closure term is done differently, as in the two rules:

floating-point-number ::=
$[sign]\ \{digit\}^{+}\ [decimal\text{-}point\ \{digit\}^{*}\ ]\ exponent$ (1)
floating-point-number ::=
$[sign]\ \{digit\}^{+}\ [decimal\text{-}point]\ \{digit\}^{*}\ exponent$ (2)

The complexity assignment for each example is based on the number of productions involved in generating it. Thus, if rule (1) is used instead of rule (2), the examples would be presented in the order shown in Fig. 7. Because +30F2 seems to be less complex than -72.S23, the user may highlight +30F2 and indicate that it is not in the expected sequence. Since the examples are valid and are otherwise sequenced correctly, the system can infer from the discourse plan that the difference between the specification and the expected sequence of examples must be caused by the bracketing of the *[decimal-point]* $\{digit\}^{*}$ component. The system can generate further examples to verify this hypothesis with the domain expert.

This illustrates how the sequencing of the examples may help detect a problem even when all of the examples and their associated prompts are valid. This is an area for future work. We must examine other domains to determine whether the KB inconsistencies that are identified via incorrect presentation sequences would typically also be manifest by either incorrect examples or incorrect prompts.

In cases where the expert selects more than one example as being faulty, the system examines the productions that were used in generating the faulty examples. If the productions have no terms in common, reasoning about each example is done independently, since the problems were probably due to entirely different reasons. Otherwise, the system engages in a clarification sub-dialogue for each common term.

The System: Implementation and Evaluation

The current system has been implemented using an NL generation system that reasons about and generates examples (Mittal & Paris 1993); Loom was used as the underlying knowledge representation system to implement the classification capabilities needed to determine the critical and variable features. The code for reasoning about possible inconsistencies was based on an assumption based truth maintenance system by Forbus and deKleer (1993). Finally, the user interface was implemented using the Common Lisp Interface Manager (CLIM). The system has thus far been used on BNF representations of various domains. However, as noted previously, the BNF notation is flexible enough to represent a large variety of domains ranging from mechanical design to protein structure.

We have not yet had an opportunity to empirically evaluate the system. In an informal study with 16 subjects, we focused on being able to find and pinpoint errors in KB specifications. We found that in almost all cases, all the users were able to detect *incorrect examples* such as the ones shown here (e.g., the floating point zero), while only 2 of the users were able to find the corresponding errors by scrutinizing the (abstract) BNF definitions for 5 minutes. Similar results were found for cases where the examples were accompanied by *incorrect prompts*.

Our test users had a much more difficult time detecting KB problems that manifested themselves as *sequencing problems* in the presentation of examples. This may be due to: (1) finding problems in a small region (just the example, or the example and the accompanying prompt) is much easier than finding problems across a larger region (finding problems in a sequence requires understanding the implications of all the examples in the sequence); (2) naturally occurring explanations are not always written in order of increasing difficulty because of other pragmatic factors (for instance, descriptions are often constrained by convention, they may be task based, etc.). Users are therefore apt to overlook this source of errors unless specifically trained to do so. In the example shown in the paper, 10 of the 16 users did not find the problematic example in Fig. 7.

These observations, while preliminary, suggest that such an interface can be very helpful in finding certain types of KB errors. It is clear that a more extensive and controlled evaluation is necessary before the actual value of such an interface can be determined. We hope to be able to conduct such an evaluation in the future, when we extend and evaluate

the system with a set of much larger KBs in various domains that have been developed as part of other projects. Note that this approach to specification debugging is most effective in complex domains, where the specifications are abstract and concept specifications are highly interrelated. Domains that are characterized by a collection of simpler rules, such as "Ships cannot berth in ports less than X feet deep" may not benefit as much from this approach. For these domains, a purely textual description of the underlying KB structures, as in the EXPECT project (Gil 1994) may be equally effective.

The methods used in this paper can be easily extended to other domains. By using the BNF notation for representing the specifications, it is clear that, at the very least, domains that can be represented using BNF-like notation can be used with this framework. This approach scales well if the domain is represented using hierarchical relationships since the system can generate text and examples focused at higher, more abstract levels; thus any sub-concepts below this level are assumed correct unless indicated otherwise.

Conclusions

The verification of the accuracy of domain representation in large KBs is a difficult problem. A visual inspection of complex terms, abstract definitions and their inter-relationships may miss some of the more intricate boundary problems in the representation. This paper has presented one approach to alleviating this problem. The scenarios presented in the paper illustrate how small mistakes in the abstract specification can be difficult to see, but can be detected by using suitable examples. Based on the discourse plan underlying the presentation, the system attempts to localize the problem in the specification.

An important advantage of this approach, as compared to previous work on example based debugging (Shapiro 1983; Mitchell, Utgoff, & Banerji 1983) is the use of the goal structure in the discourse plan to localize the possible problems in the KB. Just indicating whether an example is correct or incorrect does *not* give as much leverage as being able to state that a specific example *in a series of other, coordinated, correct examples* is wrong. Another advantage of this approach is that it allows the system to address the issue of examples that only *look* correct (syntactically correct examples generated from a faulty specification for the wrong reasons–the reasons being indicated by prompts). Finally, the point-and-click interface does not require the domain expert to be an expert in the knowledge representation language.

The work described in this paper has focused on the use of examples in describing concepts, rather than relations or processes. The acquisition and representation of knowledge for these two categories using examples is much more complex and an area for future work.

References

Ashley, K., and Aleven, V. 1992. Generating dialectical examples automatically. In *Proceedings of AAAI-92*, 654–660. San Jose, CA.: AAAI.

Carnine, D. W., and Becker, W. C. 1982. Theory of Instruction: Generalisation Issues. *Educational Psychology* 2(3–4):249–262.

Elhadad, M., and Robin, J. 1992. Controlling content realization with functional unification grammars. In *Proc. 6th Int. Wkshp on NLG*. Springer Verlag.

Forbus, K. D., and de Kleer, J. 1993. *Building problem solvers*. Cambridge, MA: MIT Press.

George, D. G.; Mewes, H. W.; and Kihara, H. 1987. A standardized format for sequence data exchange. *Protein Sequence and Data Analysis* 1(1):27–39.

Gil, Y. 1994. Knowledge refinement in a reflective architecture. In *Proceedings of AAAI-94*, 520–526, Seattle, WA: AAAI Press.

MacGregor, R. M. 1994. A description classifier for the predicate calculus. In *Proceedings of AAAI-94*, 213–220, Seattle, WA: AAAI Press.

Mitchell, T. M.; Utgoff, P. E.; and Banerji, R. 1983. Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics. In *Machine Learning: An AI Approach*. CA: Tioga Publishing Co.

Mittal, V. O., and Paris, C. L. 1993. Automatic Documentation Generation: The Interaction between Text and Examples. In *Proceedings of IJCAI-93*, 1158–1163. France.

Mittal, V. O., and Paris, C. L. 1994. Generating Examples For Use in Tutorial Explanations. In *Proceedings of ECAI-94*, 530–534. Amsterdam: John Wiley and Sons.

Mohd-Hashim, F.; Juster, N. P.; and de Pennington, A. 1994. A functional approach to redesign. *Engineering with Computers* 10(3):125–139.

Moore, J. D., and Paris, C. L. 1993. Planning Text for Advisory Dialogues: Capturing Intentional and Rhetorical Information. *Computational Linguistics* 19(4):651–694.

Musen, M. A.; Fagan, L. M.; Combs, D. M.; and Shortliffe, E. H. 1988. Use of a Domain Model to Drive an Interactive Knowledge Editing Tool. *Int. Journal of Man-Machine Studies* 26:105–121.

Pirolli, P. 1991. Effects of Examples and Their Explanations in a Lesson on Recursion: A Production System Analysis. *Cognition and Instruction* 8(3):207–259.

Reder, L. M.; Charney, D. H.; and Morgan, K. I. 1986. The Role of Elaborations in learning a skill from an Instructional Text. *Memory and Cognition* 14(1):64–78.

Reisner, P. 1981. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering* SE-7(2):229–240.

Shapiro, E. Y. 1983. *Algorithmic Program Debugging*. Cambridge, MA: The MIT Press.

Steele Jr., G. L. 1984. *Common Lisp: The Language*. Digital Press.

Swartout, W. R. 1983. The GIST Behavior Explainer. In *Proceedings of AAAI-83*. Washington, D.C.: AAAI.

Woods, W. A., and Schmolze, J. G. 1992. The KL-ONE family. *Computers and Math with Applications* 23(2-5):133–177. (Special issue on Semantic Networks in AI.)