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 inter-relationships 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 & Alevens 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...
The concept into one of two classes: in localizing problems the system categorizes each feature of plan based natural language generation. Moore and Paris (1993) for a more detailed description of generation system is beyond the scope of this paper – see examples (Mittal & Paris 1994). A complete description of the based knowledge representation systems to generate the ex- tem uses a subsumption classifier, such as the one in hadad & Robin 1992), to produce the surface form. The sys- tem interface, which converts it into a form suitable for input statements. This discourse plan is then passed to a gram- practical 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.

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

---
\(^2\)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.
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). 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 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 modified to a kleene closure as follows:

\[
\text{floating-point-number ::=}
\]

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

**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 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 (1b). The system examines the portion of the discourse plan regarding examples generated from rule (1b). Since one of the differences between the wrong example and its immediate neighbor is the exponent (the \(\{\text{decimal-point} \{\text{digit}\}^*\}\) 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:

\[
\text{exponent ::= exponent-marker [sign] \{digit\}^*} \times
\]

\[
\text{exponent ::= exponent-marker [sign] \{digit\}^+} \sqrt
\]

**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.
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 mainte-
nance system by Forbus and deKleer (1993). Finally, the
user interface was implemented using the Common Lisp In-
terface 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 re-
present a large variety of domains ranging from mechanical
design to protein structure.

We have not yet had an opportunity to empirically eval-
uate the system. In an informal study with 16 subjects, we
focused on being able to find and pinpoint errors in KB spe-
cifications. 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 (ab-
stract) 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 prob-
lems in the presentation of examples. This may be due to: (1)
finding problems in a small region (just the example, or the
element 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
elements in the sequence); (2) naturally occurring explana-
tions 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 evalua-
tion 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 sys-
tem 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 com-
plex 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 pa-
per 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 pre-
vention, 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 struc-
ture in the discourse plan to localize the possible problems in
the KB. Just indicating whether an example is correct or in-
correct does not give as much leverage as being able to state
that a specific example in a series of other, coordinated, cor-
rect 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 gen-
erated 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 pro-
cesses. 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
660. San Jose, CA.: AAAI.

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

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

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.


