GIST English Generator

Bill Swartout

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291

Abstract

This paper describes a prototype English generator which can produce English descriptions of program specifications written in Gist, a program specification language being developed at ISI. Such a facility is required because although Gist is a high level specification language, specifications written in it, like those in all other formal specification languages, are unreadable. There are several reasons for this unreadability: strange syntax; redundancy elimination; lack of thematic structure; implicit remote interactions; no representation of the motivation or rationale behind the specification; and a strict reliance on textual presentation. The current generator deals with the first two problems and part of the third. Our plans for dealing with the rest are outlined after a description of the current generator.

1. Introduction

Gist has been extensively described elsewhere [1, 2]; only a brief overview will be given here. Gist attempts to ease the burden of software design and maintenance by allowing the user to indicate what behavior he wants without requiring him to state how it is to be achieved. Gist frees its users from typical implementation concerns such as computational efficiency, method selection, and explicit data structure choice. We are exploring ways of producing efficient programs from Gist specifications by using a library of transformations which map out the specification freedoms Gist permits [6]. It is envisioned that the user will select the transformations to apply and the computer will apply them and keep a record of the applications. We expect that systems developed in this fashion will be easier to maintain because it is the specification that will be modified, rather than the highly tuned and optimized implementation code. Furthermore, the computer record of the original development will not only document that development, but also allow substantial portions of the original development to be recovered and reused after the modifications are made, thereby substantially speeding up the re-development of a working system.

This research is supported by the Air Force Systems Command, Rome Air Development Center under contract No. F30602 81 K 0056. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

2. Gist English Generator

Given a Gist specification and a small amount of additional grammatical information needed for translation (detailed below), the Gist English Generator produces an English description of the specification. There are several reasons why such a capability is important for Gist or any specification language. First, since a specification is often used as a "contract" between a customer and an implementor, it is important that all those concerned be able to understand the specification. Since customers will frequently be unfamiliar with the formal specification language, a capability for making such formal specifications understandable is needed. Second, an English translation capability can provide an alternate view of a formal specification and, hence, be useful as a debugging aid even for those familiar with the formalism such as the specifier himself. Although the English generator has been operational for only a short time, it has already made several specification errors more apparent to us. Third, since good high level specification languages embody constructs and make default assumptions that are unfamiliar to those trained to use traditional programming languages, an English generator can serve as a pedagogical aid by re-casting a specification in English, thereby shortening the time required for familiarization with both the specification language and specifications written in it.

A goal for the generator was to have it produce English directly from the specification as much as possible, as opposed to requiring the specification writer to supply substantial amounts of additional information about how the specification should be translated into English. Achieving this goal has two major benefits. First, the translation will depend more on the specification itself than on separate information required for translation, thus there is more certainty that changes in the specification will appear in the translation. Second, only minimal additional effort will be required on the part of the specification writer for his specifications to be translated into English. Given the generally dismal history of program documentation, this is an important consideration in assuring that the generator can be used with most specifications.

We recognized that to achieve this goal, specifications would have to be written following a certain style. To aid us in defining this style and to assure that it be as natural as possible, we examined existing Gist specifications to determine how the constructs of Gist were being used, and what were appropriate English translations for them. We found that most Gist forms could be mapped into English using the information supplied by the specification alone, but that relations (particularly attribute relations) and action declarations were used in several ways with differing English translations (described below). The generator uses heuristics to attempt to determine what translation should be employed for these constructs. When the heuristics are insufficient, the user can indicate the proper translation by providing additional information with the specification.

2.1. Generator Organization

The Gist English generator uses three passes and an intermediate case grammar representation [4]. The first pass of the generator examines the Gist specification and creates a case grammar representation of the English to be produced. The second pass performs inter-sentence transformations on the case grammar representation to improve the quality of the English description that will be produced. For example, this pass conjoins sentences where possible to reduce the wordiness of the explanation. The third pass uses the transformed case grammar to produce actual English. The third pass also performs some intra-sentence optimizations. One such transformation was motivated by the observation that sentences which mention a definite reference first followed by an indefinite are clearer than those in which the two are reversed. For example, "A pier of the manager's port must be the pier p" is not as clear as "The pier p must be a pier of the manager's port".

This organization has some distinct advantages. The multiple representations provide appropriate points for making transformations. For example, transformations which are primarily concerned with English, such as conjunction insertion, are most appropriately made on the case grammar. It would be much more awkward to make such transformations during the first pass. Another advantage is that the case grammar and passes two and three of the generator are independent of the particular representation used for Gist specifications. Thus, the portions of the generator that embody English knowledge can be exported to other applications.

Text generation has been investigated by a number of researchers (see [7] for a current bibliography). Boris Katz has produced a generator which is perhaps most similarly structured to the one presented here [5]. He has concentrated more on pass two, that is, on transforming the primitive English description formed in pass one into a more fluent explanation. In the Gist generator, the most difficult task has been forming the primitive explanation, since Gist and English are often quite different in terms of what they can represent easily.

2.2. Translating Attribute Relations

Attribute relations and action declarations require a richer set of translations than other Gist constructs. This section and the next outline how their English translation is performed.

Attribute relations are binary relations declared as part of type declarations. For example, the type declaration:

```
type ship (Destination | port);
```

declares the type ship and a relation Destination between ships and ports. When used elsewhere in Gist, :Destination is used to indicate the mapping from ships to ports, and ::Destination is used for the back-mapping from ports to ships. For example,

```
(1) ship:Destination
```

refers to the destination of a ship, and

```
(2) port::Destination
```

refers to a ship whose destination is the port.

2.2.1. Kinds of Attribute Relations

We have identified three major uses of attribute relations which have different English translations. The first use is illustrated by the example given above. To translate the forward mapping, the relation name may be used as a noun modified by a genitive form of the declared type. Thus (1) above translates as "the destination of the ship" or "the ship's destination". To translate the back-mapping we generate a noun phrase whose head noun is the type of object being referred to (in this case, ship) modified by a relative clause indicating the relationship. Thus, (2) translates as "the ship whose destination is the port". In the absence of other information, this kind of translation is employed by the generator.

Attribute relations are also used to indicate "part of" relations. In the specifications we examined, this type of relationship was usually indicated by giving the relation the same name as the attribute type, as in:

```
type auto (Engine | engine);
```

This declares that every object of type auto has an attribute called Engine, whose type is engine. The translation for part-of relations is similar to those described above, but the verb "have" or "belong" is used in place of "is". Thus, the type declaration itself would translate as "Each auto has an engine", and forward mappings auto:Engine from autos to engines would translate as "the auto's engine", while backmappings auto::Engine from engines to autos would translate as "the auto that has the engine". The "part of" translation is used by the generator whenever the name of the relationship is the same as its attribute type.

Finally, attribute relations are sometimes used as verbs, as in:

```
type pier (Handle | cargo);
```

The translation for this type declaration is "Each pier handles a cargo". The type being declared is taken to be the subject and the attribute type is the object.

```
pier:Handle
```

translates as "The cargo which is handled by the pier", and

```
cargo::Handle
```

translates as "The pier which handles the cargo". One problem with the verb form is that the translation can be very awkward if it appears deeply embedded within other forms. The generator recognizes such situations and uses several sentences to describe these embedded forms, automatically introducing intermediaries as needed (an example appears below). The specifier must indicate that an attribute relation is to be translated as a verb by placing a verb property on the name of the relation. This is one of the small grammatical additions required for natural language generation.

2.3. Translating Actions

Actions correspond to verbs. In the specifications we have encountered so far, the name given to an action is either an English verb, or it is a compound name (e.g. MoveShip or Replace__Line) where the first element of the compound corresponds to the verb. If a name is not a compound, the generator assumes that it is an English verb. If it is a compound,* the generator assumes that the first element of the compound is the verb.

*The generator knows about stylized ways of creating compounds, including separating by upper and lower case, hyphens and underscores, and it can break these compounds apart.

The parameters in an action correspond to cases in a case grammar. The generator knows how to map a fixed set of cases into English. The user must supply the generator with annotations for each action declaration telling it the action's parameter/case correspondences. This is the second (and final) grammatical addition required for natural language generation.

Currently the generator allows six cases for parameters. We expect this number to grow slightly as we gain more experience with a wider variety of specifications. The current cases include:

- Agent, the thing or person performing the action. Becomes the subject of declarative sentences. (The manager moves the ship.)
- Object, the thing or person upon which the action is performed. Becomes the subject of passive sentences. (John kicked the ball.)
- Instrument, the thing used to perform the action. When translated to English, an Instrument is preceded by the preposition "with". (I dug the hole with a shovel.)
- Dative, corresponds to the indirect object. When translated, the Dative case is preceded by "to". (I gave the ball to the boy.)
- Directional, indicates the object toward which the action is proceeding. This case is also preceded by "to". (Move the ship to the pier.)
- Locative, nouns in this case indicate the location of the action. The user supplies the appropriate preposition to be used with this case. (The ship sank at the pier.)

For example, to translate the action:

```
action MoveShip[s | ship, p | pier]
```

The user would inform the generator that the first parameter was the Object and the second was the Directional. The action would then translate as:

"Move the ship s to the pier p."

2.4. Examples

This section presents some examples of English produced by the generator. The first example is a preliminary specification for a harbor manager. (The reader is not expected to understand the Gist specifications before reading their English paraphrases.)

```
begin
 type port(Pier | pier ::unique);
 type pier(Handle | cargo, Slip | slip ::unique);
  type slip();
  type ship(Carry | cargo, Destination | port,
        berth | slip :optional ::optional);
  type cargo();
 agent manager(Port | port :unique ::unique)
  where action MoveShip[s | ship, p | pier]
         precondition s :berth ::Slip ::Pier=
                                manager :Port
         precondition manager :Port :Pier=p
         definition update :berth of s to p:Slip;
        action LoadShip[s | ship, c | cargo]
          precondition s :berth ::Slip :Handle=c
          precondition s :berth ::Slip ::Pier=
                                  manager :Port
          definition insert s :Carry=c;
        action AssignCargo[c | cargo, p | port]
          definition LoadShip[p ::Destination, c]
 end
end
```

To create an English description for this spec, the specifier had to inform the generator that the attribute relations Carry and Handle should be translated as verbs, and he had to indicate the appropriate cases for each of the parameters of the action declarations. The English that resulted appears below:

> There are ports, ships, cargos and managers.
>
> Each port has one pier. Each pier belongs to one port.
>
> Each pier handles one cargo and has one slip. Each slip belongs to one pier.
>
> Each ship carries one cargo, has one destination which is a port and may have a berth which is a slip. A slip optionally is the berth of a ship.
>
> Each manager has one port. Each port belongs to one manager. A manager can assign a cargo, load a ship or move a ship.
>
> To move a ship s to a pier p:
>
> Action: The berth of the ship s is updated to the slip of the pier p.
>
> Preconditions: The pier of the manager´s port must be the pier p. The slip of the pier of the manager´s port must be the berth of the ship s.
>
> To load a cargo c on a ship s:
>
> Action: Assert: The ship s carries the cargo c.
>
> Preconditions: The slip of the pier of the manager´s port must be the berth of the ship s. The pier that has the berth of the ship s must handle the cargo c.
>
> To assign a cargo c to a port p:
>
> Action: Load the cargo c on a ship whose destination is the port p.

When the person who wrote this specification saw the English description of it, he immediately realized that he had made a mistake, because ports should have more than one pier and piers should have more than one slip. This mistake had been hidden in the Gist spec because Gist defaults the mapping for an attribute relation to unique in the forward direction (i.e. from ports to piers and piers to slips). After correcting those bugs and making some additions, a new specification and English description were produced:

```
begin
 type port(Pier | pier :multiple ::unique,
         harbor | ship :any ::optional);
 type pier(Handle | cargo :multiple,
         Slip | slip :multiple ::unique);
 type slip();
 type ship(Carry | cargo :any,Destination | port,
         berth | slip :optional ::optional);
 always required Berths__Are__In__Ports
  for all s | ship ||
   s :berth=$=>s ::harbor :Pier :Slip=s : berth;
 type cargo()optional
              supertype of<grain();
                           fuel()> ;
 always prohibited Fuel__And__Grain
    there exists s | ship, g | grain, f | fuel ||
                   s :Carry=g and s :Carry=f ;
 agent manager(Port | port :unique ::unique)
   where action MoveShip[s | ship, p | pier]
           precondition s ::harbor=manager :Port
           precondition manager :Port :Pier=p
           definition update :berth of s to
                        p :Slip;
         action LoadShip[s | ship, c | cargo]
           precondition s:berth::Slip:Handle=c
           precondition s ::harbor=manager :Port
           definition insert s :Carry=c;
         action AssignCargo[c | cargo, p | port]
           definition LoadShip[p::Destination,c]
   end
end
```

The English description for the above spec:
(Comments appear underlined.)
(The generator "sets the stage" by first creating a summary statement of the top-level types that will be described. Top-level types are those that are not either subtypes or part-of some other type.)
There are ports, ships, cargos and managers.

Each port has multiple piers. Each pier belongs to one port. Each port harbors any number of ships. Each ship may be harbored by a port.
(For each type, the generator constructs a description of its attribute relations. Note the change from the previous spec.)
Each pier handles multiple cargos and has multiple slips. Each slip belongs to one pier.

Each ship carries any number of cargos, has one destination which is a port and may have a berth which is a slip. A slip optionally is the berth of a ship.

(Wherever possible, the second pass of the generator conjoins sentences. Before pass 2, the above paragraph contained four sentences. After pass 2 the first three have been conjoined into one.)
Fuels and grains are cargos.

Each manager has one port. Each port belongs to one manager. A manager can move a ship, load a ship or assign a cargo.
(The generator gives a summary description of the actions an agent can perform before describing them in detail.)

To move a ship s to a pier p:
 Action: The berth of the ship s is updated to a slip of the pier p.

 Preconditions: The pier p must be a pier of the manager´s port. The manager´s port must harbor the ship s.

To load a cargo c on a ship s:
 Action: Assert: The ship s carries the cargo c.

 Preconditions: The manager´s port must harbor the ship s. The pier that has the berth of the ship s must handle the cargo c.

To assign a cargo c to a port p:
 Action: Load the cargo c on a ship whose destination is the port p.
(In describing actions, preconditions are described after the actions, because they represent a more detailed level of description than the action itself.)
Fuel And Grain:
 A ship s must not carry a grain g and a fuel f.

Berths Are In Ports:
 If: A ship s has any berth,
 Then: A port p harbors the ship s. The berth of the ship s is a slip of a pier of p.
(Since this global constraint embedded the verb attribute relation "Harbor", the generator split the description up into multiple sentences and introduced the intermediary "p" to make the description clearer.)

3. Current Research Issues

While the generation capability described above has already demonstrated its usefulness in making Gist specifications more readable, there is much that can be done to improve it. There are four topics we are currently investigating which we expect will substantially improve the quality of the explanations that can be offered. These are: global explanation descriptions, presentational form, level of abstraction, and symbolic execution.

One problem with the current English generator is that it makes its decisions based almost entirely on local information. That is, when translating a piece of a specification to English, decisions about how that translation should be made depend just on the particular piece of specification. Operations such as user modelling, choosing appropriate names for objects, and producing focused explanations which describe a subpart of the specification in relation to the rest all require a more global view of the explanation: the explanation itself must be viewed as a whole and manipulated before being presented. Just as the use of a case grammar provides the English generator with an intermediate representation which is more appropriate for operations such as conjunction insertion that require a more global view than surface syntax provides, a global explanation description is required for the kinds of operations mentioned above.

Currently, explanations are only available in one presentational form: English text. Yet text is often not the clearest way of presenting an explanation. For example, most machine-produced English explanations of highly interconnected structures (such as a causal network) become rapidly confusing. The same information is substantially clearer when illustrated by a drawing. This suggests that an explainer will benefit from an ability to inter-mix multiple presentation forms, choosing the most appropriate one given the nature of the information to be presented and knowledge of the capabilities and preferences of the user. A preliminary graphic capability has been designed for Gist and is currently being implemented. This will allow the user to display, enter, and modify some of the information in a Gist specification. The next stage will be to integrate this capability with the English generator and a set of heuristics for choosing the most appropriate form, so that the explainer will be able to integrate graphic and text explanations.

It is generally agreed that to give good explanations, it is necessary to be able to summarize the information to be presented so that the listener is not overwhelmed by detail. The current generator has a limited ability to summarize. For example, the actions an agent can perform are presented in an overview before they are described in detail. While such Gist-based heuristics can be valuable, they will probably not be powerful enough to solve the summarization problem by themselves. The problem is that such heuristics only examine the final version of the specification and frequently, there is not sufficient information available to determine appropriate summarizations. A record of how the specification itself was developed would be very valuable, because it could detail how the final specification was elaborated from a more abstract initial specification and give the rationale behind those elaborations. This record could be used both in determining summarizations and in justifying the specification. The Gist language itself has no special features for representing these different levels of abstraction. We are currently designing a system for incrementally acquiring specifications from the specification writer. This system will allow the writer to initially give a very high-level, abstract specification. This initial description will usually be incomplete. The initial specification will be repeatedly elaborated until it is as detailed as required. This process will be recorded. The record should give the explainer a needed additional source of knowledge for providing good explanations.

Finally, we wish to be able to explain the dynamic behavior implied by a specification. Often, subtle aspects of a program's (or specification's) behavior only become apparent by executing it. A symbolic evaluator has been designed and partially implemented by Don Cohen [3] which allows the user to evaluate the specification with symbolic inputs, rather than with specific concrete inputs (although the user may supply concrete inputs if desired). In this way, the user can test a specification on a whole class of inputs at once, rather than laboriously iterating over all the possible instances of that class. While the general problem of symbolic evaluation is very hard (complex loops, for example, present behaviors that can be very difficult to express in closed form) we have found that most of the forms which actually arise in specifications are relatively easy to deal with. Those few forms that present problems can then be evaluated concretely. The output of this evaluator is a trace which characterizes the implied behaviors of the specification. The trace is unreadable by people because it is too detailed and unfocused. A dynamic explanation capability is being designed to translate this trace into a more readable form.

REFERENCES

1. Balzer, R. M., and N. M. Goldman. Principles of good software specification and their implications for specification languages. Proceedings of the Specifications of Reliable Software Conference, Boston, Massachusetts, April, 1979, pp. 58-67. (Also presented at the National Computer Conference, 1981.)

2. Balzer, R., Goldman, N. & Wile, D. Operational specification as the basis for rapid prototyping. Proceedings of the Second Software Engineering Symposium: Workshop on Rapid Prototyping, ACM SIGSOFT, April, 1982.

3. Cohen, D., Swartout, W. & Balzer, R. Using symbolic execution to characterize behavior. Proceedings of the Second Software Engineering Symposium: Workshop on Rapid Prototyping, ACM SIGSOFT, April, 1982.

4. Fillmore, C. The Case for Case. In Universals in Linguistic Theory, Holt, Rinehart and Winston, 1968.

5. Katz, B. A Three-Step Procedure for Language Generation. Tech. Rept. AI Memo 599, MIT, December, 1980.

6. London, P. & Feather, M.S. Implementing specification freedoms. Tech. Rept. RR-81-100, ISI, 4676 Admiralty Way, Marina del Rey, CA 90291, 1981. Submitted to Science of Computer Programming.

7. Mann, W.C.,M. Bates,B. Grosz, D. McDonald,K. McKeown, W. Swartout. Text Generation: The State of the Art and the Literature. Tech. Rept. RR-81-101, ISI, December, 1981.