# The GIST Behavior Explainer

Bill Swartout

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

## Abstract

One difficulty in understanding formal specifications is that there are often interactions between pieces of the specification, never explicitly stated, that only become apparent when the specification is analyzed or simulated. Symbolic evaluation has been proposed as a way of making such interactions apparent, but symbolic evaluators often produce enormous execution traces that are tedious and difficult to examine. This paper presents an automated system that employs a number of heuristics to select the most interesting aspects of the trace for presentation. The system uses this information to construct an English description of the trace. Due to the need for summarization and proof reformulation, the direct-translation approach, which worked well in describing specifications statically, is not suitable in this case. This paper describes the system and gives an example of its output.

## 1. Introduction

Regardless of the specification language used, formal program specifications can be tough to understand. Yet, because a specification is frequently the means by which a customer communicates his desires to a programmer, it is critical that both the customer and programmer be able to examine and comprehend the specification. Our experience with Gist, a high-level specification language being developed at ISI [1], has indicated that two of the major impediments to understandability are the unfamiliar syntactic constructs of the language and non-local interactions between parts of the specification. These interactions are often not apparent from a casual examination of the specification.

In an earlier paper, the Gist paraphraser and English generator were described [6]. These address the syntax problem by directly translating a Gist specification into English. We have found the paraphraser to be useful in both clarifying specifications and revealing specification errors. We expected that the English translation would be useful to people unfamiliar with Gist, because it would make Gist specifications accessible, but we were surprised to discover that experienced Gist users found it helpful for locating errors. The reason is that an English translation gives the specifier an alternate view of his specification which highlights some aspects of the specification which are easily overlooked in the formal Gist notation. But the paraphraser deals only with the static aspects of a specification. This paper deals with the more difficult problem of making non-local specification interactions apparent by simulating the dynamic behavior implied by the specification.

Our approach has been to discover non-local interactions by using a symbolic evaluator, developed by Don Cohen [2], to analyze a specification. As it evaluates the specification, the symbolic evaluator creates a description of the relationships among pieces of the specification. It discovers what sorts of behaviors the specification allows, and what is prohibited by constraints. A symbolic evaluator does not require specific inputs. Instead it develops a description of the range of possible responses to a given range of inputs. Due to this characteristic, it is possible to test a specification symbolically over a range of inputs that would require many test runs if specific inputs were employed.

A specifier interested in the behavior of his specification may direct the evaluator to execute one of the actions defined in the specification. As the evaluator executes the action, some apparently possible execution paths may be eliminated due to constraints, and a more detailed description of the inter-relationships within the specification is developed.

The symbolic evaluator produces an execution trace, which details everything discovered about the specification during evaluation. The trace includes not only base facts directly implied by the specification, but also any further implications that the evaluator may have derived from the base facts using its theorem prover. In addition, the trace records the proof structures justifying the facts it contains. Unfortunately, the trace is much too detailed and low-level to be readily understood by most people. To overcome that difficulty, we have constructed a trace explainer that selects from the trace those aspects believed to be interesting or surprising to the user and uses that information to produce an English summary.

There are a number of problems that make the simple direct-translation techniques (which worked well for the Gist paraphraser) unsuitable for the trace explainer. These problems include:

> **Detail suppression.** The trace is much too detailed to be described in its entirety. The trace explainer uses the structure of the specification and heuristics about what the user is likely to find interesting or surprising in selecting what to describe.

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. I wish to thank Robert Balzer, Don Cohen, Neil Goldman, Jack Mostow and Dave Wile for their comments and discussions.

- **Proof summarization and reformulation.** The symbolic evaluator uses an augmented resolution-based theorem prover in deriving the consequences of the specification. While this approach is arguably attractive for its generality and simplicity, its arcane proof structures could impose a hardship on the user. The trace explainer attempts to reformulate resolution proof structures into more familiar and understandable ones.

- **Referring expressions.** With the Gist paraphraser, it was usually acceptable to use the name given to an object in the specification as its referring expression in the English paraphrase. The trace explainer cannot rely on this technique alone, since there are objects in the trace that do not appear in the specification. Moreover, depending on context, different referring phrases may be necessary even though the same object is being referred to.

The next section presents an example specification and a machine-produced description of its symbolic evaluation. Following that, the example will be used to illustrate the initial solutions we have found for the problems listed above. The final section outlines some of the further work needed to extend the capabilities of the explainer.

## 2. An Example

The example presented here is a simplified version of a specification for a postal package router (see [3, 5]). The package router is designed to sort packages into bins corresponding to their destinations. A package arrives at a location called the

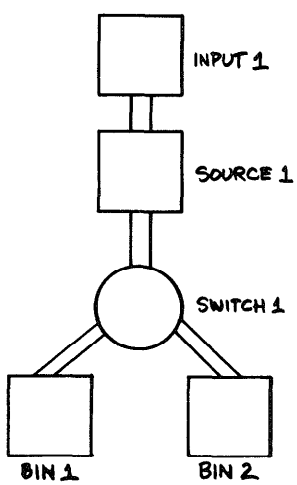


**Figure 2-1:** Package Router

*source* and its destination is read there. A binary tree of switches and pipes connects the source with the output bins. It is the job of the package router to set the switches so that the package winds up in the proper destination bin (see Figure 2-1). The simplified specification contains just one switch and two bins. In addition, a location called the *input* has been defined, which is where all boxes are originally located. The formal Gist specification appears in Figure 2-2. It is not necessary to understand the formal notations, since an English translation of the specification (produced by the paraphraser) is available: Figure 2-3 is the English paraphrase of the specification's type structure and Figure 2-4 describes the possible actions in this specification.

Having defined the type structure and actions, a specifier may wish to define some test sequences of actions to see how the constraints of the specification interact to limit the behavior of the specification in ways that are not obvious from the static specification alone. In Figure 2-5, the user has defined such a test sequence. The user has also given preconditions to define the initial state and the structure of the switching network and a postcondition to describe the final goal of the system. Notice that

```
begin
  type box(Location | location, Destination |
                                      bin);
  type location()unique supertype of
      <input() definition{input1};
       source(Source-outlet | switch)
              definition{Source1};
       internal-location()unique supertype of
          <switch
           (Selected-outlet
                   |internal-location,
            Outlet|internal-location
                        :multiple)
                 definition{switch1};
           bin() definition{bin1, bin2}>>;
  agent PackageRouter() where
       action Insert[box]
         definition update :Location of box
                      from input1 to Source1;
       action Set[switch]
         precondition ~$:Location=switch
         definition update :Selected-outlet
                of switch to switch :Outlet;
       action Move[box]
         precondition box:Location=Source1 or
                      box :Location=a switch
         definition
             if box :Location=Source1
                then update :Location of box
                     to Source1:Source-outlet
                else update :Location of box
                     to box :Location
                          :Selected-outlet;
       action Test[]
         precondition switch1 :Outlet=bin1
         precondition switch1 :Outlet=bin2
         precondition Source1 :Source-outlet=
                                 switch1
         precondition for all box ||
              box :Location=input1
         postcondition for all box ||
              box :Location=box :Destination
         definition begin
                      Insert[a box];
                      Move[a box];
                      Insert[a box];
                      Move[a box];
                      Set[a switch];
                      Move[a box];
                      Move[a box]
                    end
    end
end
```

**Figure 2-2:** Formal Gist Specification for Package Router

There are boxes, locations and package-routers.

Each box has one location. Each box has one destination which is a bin.

Internal-locations, sources and inputs are locations.
Bins and switches are internal-locations.
Bin1 and bin2 are the only bins.
Switch1 is the only switch. The switch has one selected-outlet which is an internal-location. The switch has multiple outlets which are internal-locations.
Source1 is the only source. The source has one source-outlet which is a switch.
Input1 is the only input.

**Figure 2-3:** Paraphrase of Package Router Type Structure

---

A package-router can insert a box, set a switch, or move a box.
To insert a box:
Action: The box's location is updated from input1 to source1.
To set a switch:
Action: The switch's selected-outlet is updated to an outlet of the switch.
Preconditions:
The switch must not be the location of any box.
To move a box:
Action:
If: The box's location is source1,
Then: The box's location is updated to the source-outlet of source1.
Else: The box's location is updated to the selected-outlet of the switch that is the box's location.
Preconditions:
Either:
1. The box's location must be source1, or
2. The box's location must be a switch.

**Figure 2-4:** English Paraphrase of Possible Actions

in the action body, all operands are specified non-deterministically. For example, the first action invocation states that a box is to be inserted, but it does not say which box. The intent of such a statement is that any box may be inserted, as long as no constraints are violated. This non-deterministic reference is one of the freedoms allowed by the Gist specification language which gives the specifier greater expressive power and prevents him from having to over-specify behaviors. Because the user does not have to explicitly select parameters, he can see with just one test action whether it is ever possible to achieve the postconditions using the particular sequence of action invocations given.

After the symbolic evaluator runs, the specifier can use the trace explainer to see an overview of the results of symbolic execution (see Figure 2-6).

To test:
Action:
1. Insert a box.
2. Move a box.
3. Insert a box.
4. Move a box.
5. Set a switch.
6. Move a box.
7. Move a box.

Preconditions:
For all boxes:
The box's location must be input1.
The source-outlet of source1 must be switch1.
An outlet of switch1 must be bin2.
An outlet of switch1 must be bin1.
Postconditions:
For all boxes:
The box's location must be the box's destination.

**Figure 2-5:** English Paraphrase of a Test Action

---

1. A box, call it box1, is inserted.
Result: The new location of box1 is source1.
*The explainer describes the action invocation as it was stated in the test case. It makes up the name "box1" for this box so that it can be conveniently referred to later. The explainer then describes the result of this action invocation.*

2. A box is moved. The box must be box1 since
2.1 For all boxes except box1, the box's location is input1, and
2.2 The precondition of moving a box requires that either:
2.2.1 The box's location must be source1, or
2.2.2 The box's location must be a switch.
Result: The new location of box1 is switch1.
*Something surprising has happened. In the test case, the action invocation was made with a non-deterministic parameter, but the constraints of the specification force the selection of one particular box, namely box1. The explainer recognizes this sort of behavior as surprising and describes not only the restriction on binding the parameter, but also the reasons behind it.*

3. A box, call it box2, is inserted. The box must not be box1 since
3.1 The location of box1 is switch1, and
3.2 The location of the box to be inserted must be input1 since the update in inserting a box requires it.
Result: The new location of box2 is source1.

4. A box is moved. The box must be box1 since otherwise, at the start of step 5, the location of box2 would be switch1 but the precondition of setting a switch requires that the switch must not be the location of any box.
Result: The new location of box1 is the selected-outlet

**Figure 2-6:** Machine-Produced Description of Symbolic Evaluation of Test
(continued on next page)

**of switch1. Switch1 is not the location of any box.**
*At the start of step 4, box1 is at the switch, and box2 is at source1. It would appear that either one could be moved in step 4 since both satisfy the preconditions of the move. However, if box2 moved, it would be impossible to execute the next step. So, as the explainer describes, the non-local interaction with step 5 constrains the parameter binding.*

**5. A switch is set. The switch must be switch1 since there are no other switches.**
**Result: The new selected-outlet of switch1 is an outlet, call it outlet1, of the switch.**

**6. A box is moved. The box must be box2 since the precondition of moving a box requires that either:**
**6.1 The box's location must be source1, or**
**6.2 The box's location must be a switch.**
**Result: The new location of box2 is switch1.**
*The proof that the box to be moved must be box2 is actually quite involved. The system currently has no good way of summarizing proofs of this type, so it falls back on another heuristic. The explainer examines the proof structure to find the statement in the specification that was used specifically to constrain this choice and displays it. That is, rather than showing a proof, we just display the parts of the specification that became relevant in constraining this behavior. This heuristic seems to work well, and it provides the explainer with an "escape" so that it can convey some information even if it can't reformulate the proof. Although it's usually not too difficult to figure out how the specification statement constrains the behavior, we plan to add a facility to allow the user to ask for further elaborations when he as trouble (see "Future Directions").*

**7. A box is moved. The box must be box2.**
**Result: The new location of box2 is outlet1. For all boxes, the box's location is the box's destination.**
*Since the justification for this step is the same as for the preceding, the explainer omits it.*

**Figure 2-6, continued**

# 3. System Organization

Our facilities for making specifications more understandable are organized as shown in Figure 3-1. Like the Gist paraphraser, the trace explainer employs an intermediate case frame representation which is converted to English by a relatively straightforward English generator. The explainer itself is organized into individual explanation methods. There are two basic kinds of explanation methods. *Trace-based* methods can describe particular situations that arise in the trace, such as an action invocation or the justification of a fact found by the evaluator. The other kind, *structuring* methods, organize the output of the trace-based methods into higher-level explanation structures. For example, one such explanation method organizes two statements into a statement-reason explanation structure of the form "P since Q" (see [8]).

There can be several explanation methods that describe the same object or behavior, but at differing levels of detail or highlighting different aspects. It is up to the explainer to choose the most appropriate explanation method for a given situation. Currently, much of this decision-making is handled procedurally. While this organization has been adequate to handle the sorts of specifications shown here, a more sophisticated explanation planning mechanism will probably be needed to handle larger specifications.

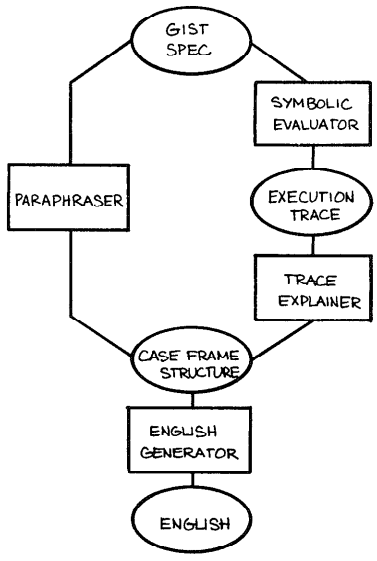


**Figure 3-1:** System Overview

# 4. Issues in Explaining the Trace

The chief problems confronting us in explaining the trace have been 1) selecting and summarizing the most appropriate information to present to the user from the large number of inferences produced by the symbolic evaluator, 2) reformulating the theorem prover's proofs into a more understandable form and 3) dealing with changing referring expressions.

## 4.1. Selection and Summarization

Both the structure of the specification and heuristics about what the user wants to see are used to guide the summarization of the trace. We assume that a particular specification has the structure it has because it models to some degree the way the specifier thought about the problem. Some of the explanation methods exploit this structure. Consider two explanation methods, both offering descriptions of action invocations. One might use the structure of the specification and produce a very summary description by just translating the invocation statement itself and stating the results of the invocation (similar to the example given above). Another explanation method could give a more detailed description by actually describing the body of the action that was invoked as well. The structure of the specification is a help in summarizing the trace, but it is not enough since many of the facts the evaluator discovers (and the explainer must chose among) come from the interaction of several pieces of the specification.

To decide which interactions to present, the system must have some idea of what the user will be interested in. For example, a customer unfamiliar with the specification might want an overview that described the "main line" or normal execution path. On the other hand, the specifier who wrote the specification would want to see the parts of the specification that appear to be incorrect because they use the specification language in a surprising or unusual way. Our current implementation has concentrated on presenting these surprising behaviors, rather than the normal case.

What, then, is surprising? We consider things such as superfluous code, the use of an overly general language construct, or, worst of all, a specification which is inherently contradictory to be surprising. More specifically, a conditional branch which must always follow the same path, constraints which are never employed, and (as in the example presented here) the use of a non-deterministic parameter that turns out to be deterministic are all surprising. The explainer's methods recognize surprising situations and describe them to the user.

The kinds of surprises described above are language-dependent. Another kind of surprising situation will arise as our work on incremental specification proceeds further. The incremental view of specification states that detailed specifications do not appear all at once, but rather are gradually refined layer by layer from more abstract specifications. Each succeeding layer is in a sense an implementation of the one above it. Surprises will occur when the symbolic evaluator discovers that one layer of a specification does not meet the goals set forth for it at a higher level.

### 4.2. Reformulating Proofs

While a resolution theorem prover may be attractive for many reasons, certainly the lucidity of its proofs is not one of them. Our approach to this problem follows that suggested by Webber and Joshi [7]: we attempt to reformulate the resolution proofs into ones that seem more natural. Some of the recognizers we have developed find simple proof structures like modus ponens, while others find more complicated structures such as proof by contradiction or a version of the pigeonhole principle. For example, the pigeonhole rule examined the proof that the box moved in step 2 is box1 and recognized that the proof has the form of successively eliminating possible candidates. Since one reformulation may cover several resolution steps, recognizers like this help both by reducing the amount of information that must be conveyed and by structuring it more appropriately.

At times the recognizers alone provide sufficient information to know how a proof should be described. At other times it is necessary to consider how the proof description fits into the trace description as a whole. For example, in describing step 4 in the example, a hypothetical construction was used:

> *otherwise, at the start of step 5, the location of box2 would be switch1*

since the selection of the box to be moved was constrained by an event still in the future.

### 4.3. Referring Expressions

Because the symbolic evaluator dynamically creates symbolic instances of types as it reasons about them, the trace explainer must be able to create names for such objects, even though they never appear in the original specification. For example, *Box1*, mentioned in line 1 of the trace description never appears in the specification. It is a symbolic instance created by the evaluator to represent "the box inserted in step 1". While the evaluator creates a new instance at each action invocation, the explainer is more parsimonious, creating new names only when equivalence to previous names cannot be established. Thus, in step 2, no new name is required to describe the box to be moved since it must be box1.

While names like *box1* or *box2* are often sufficient for naming symbolic instances, they can at times be more confusing than helpful. Consider line 3.2. *The box to be inserted* referred to there is in fact equivalent to *box2*. But substituting *box2* in place of *the box to be inserted* results in a confusing explanation. That's somewhat surprising, since one would expect that after naming an object in a description one would be free to use that name to refer to it. The problem is that the order of the description does not correspond to the ordering of events. The reasoning about which box to insert precedes its selection and naming, but in the description, things are reversed and the naming of objects is sensitive to the order of events. The explainer therefore generates the phrase *the box to be inserted* rather than *box2*.

## 5. Future Directions

Our current implementations of the symbolic evaluator and trace explainer produced the examples contained in this paper. While our systems are still very much laboratory prototypes, we feel that they have begun to demonstrate the utility of the techniques outlined here in debugging specifications. Even so, we are aware that these techniques will not, by themselves, be sufficient for much larger specifications. The four areas that seem to need attention are the symbolic evaluator, incremental specification, allowing the user to ask follow-on questions about the summaries the explainer provides, and a better mechanism for planning explanations.

The current symbolic evaluator is not goal driven. Rather than having a model of what might be interesting to look for in a specification, the evaluator basically does forward-chaining reasoning until it reaches some heuristic cut-offs. In the process, it generates interesting as well as uninteresting results, which the explainer must sift through. While this works reasonably well for the small specifications we have been working with, larger specifications could prove overwhelming. One solution may be to make the symbolic evaluator more goal-directed. By giving it, at least at a high level, a model of what might be interesting, it could be more directed in its search. After narrowing the search using goals, the evaluator could then switch to forward-chaining to more completely examine the smaller problem space. Such an approach would benefit both the evaluator because it would run faster, and the explainer, because the goal structure would aid substantially in generating explanations.

The notion of incremental specification has already been mentioned above. Aside from indicating surprising behaviors, incremental specification could also improve the performance of the evaluator through higher level abstractions [4], since a few reasoning steps at the high level could replace many low level inference steps.

The current implementation of the explainer makes no provision for the user to ask further questions about the descriptions it produces. However, such a capability is required because the descriptions are produced heuristically. The system may assume that the user will readily understand something that actually requires further description. For the near future, we do not envision allowing the user to ask questions in natural language, but instead, we will let him point at the pieces of the description he did not understand (using a mouse or other pointing device) and ask for further description.

Finally, we are currently implementing an explanation planning mechanism that will allow us to represent plans for presenting information. This mechanism will allow us to describe goals and the capabilities of plans along multi-dimensional scales. The dimensions will be either categorical or ordinal. For example, some of the kinds of dimensions that seem to be important in explanation are: the type of object to be described, the form the description is to take, degree of verbosity, and level of detail. The planning mechanism will support matching goals and methods represented in this space, and will provide a mechanism for selecting the most appropriate method when only a partial match can be found.

## References


1. 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.

2. Cohen, D. Symbolic execution of the Gist specification language. Proceedings of the Eighth IJCAI, IJCAI, 1983.

3. Hommel, G. (ed.). Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage. Kernforschungszentrum Karlsruhe GmbH, August, 1980. PDV-Report, KfK-PDV 186, Part 1

4. Sussman, G. SLICES: at the boundary between analysis and synthesis. Tech. Rept. AI Memo 433, MIT, July, 1977.

5. Swartout, W. and Balzer, R. "On the Inevitable Intertwining of Specification and Implementation." *Communications of the ACM 25*, 7 (July 1982), 438:440.

6. Swartout, W. Gist English Generator. Proceedings of AAAI-82, AAAI, 1982.

7. Webber and Joshi. Taking the initiative in natural language data base interactions: justifying why. University of Pennsylvania, 1982.

8. Weiner, J. "BLAH, A system which explains its reasoning." *Artificial Intelligence 15* (1980), 19-48.