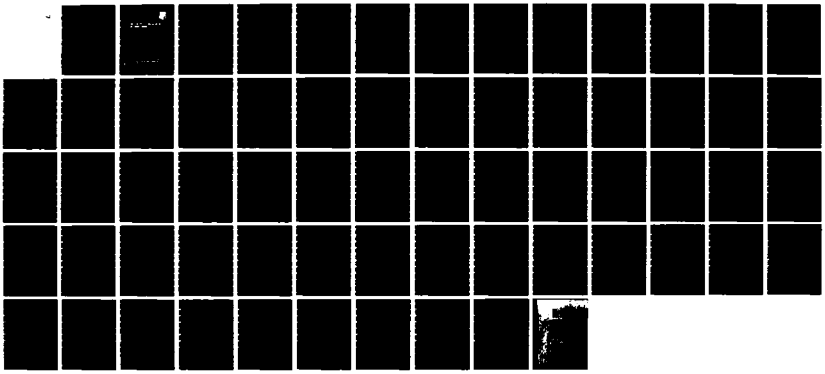
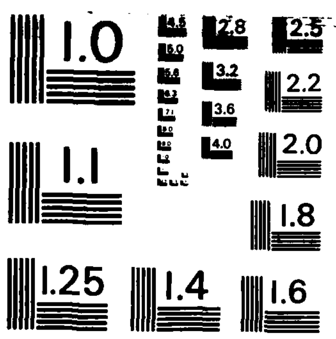
AD-A141 079 TOOLS FOR SPECIFICATION VALIDATION AND UNDERSTANDING (U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST R BALZER ET AL. DEC 83 1/1

UNCLASSIFIED RADC-TR-83-292 F30602-81-K-0056 F/G 9/2 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(12)

AD-A141 079

RADC-TR-83-292
Final Technical Report
December 1983

TOOLS FOR SPECIFICATION VALIDATION AND UNDERSTANDING

University of Southern California

Robert Balzer, Donald Cohen and William Swartout

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

84 05 15 220

RADC-TR-83-292 has been reviewed and is approved for publication.

APPROVED: [illegible]

WILLIAM E. RZEPKA
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTIZ, JR.
Acting Technical Director
Command and Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-292	2. GOVT ACCESSION NO. AD-A141 079	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) TOOLS FOR SPECIFICATION VALIDATION AND UNDERSTANDING		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 29 Jan 81 - 31 May 83
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Robert Balzer, Donald Cohen, William Swartout		8. CONTRACT OR GRANT NUMBER(s) F30602-81-K-0056
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Southern California, Information Sciences Institute, Marina del Rey CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55812207
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE December 1983
		13. NUMBER OF PAGES 62
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: William E. Rzepka (COEE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Requirements	Specification Language
Software Specification	Executable Specification Language
Symbolic Execution	Software Validation
Requirements Language	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Regardless of the specification language used, formal program specifications can be difficult 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. Experience with Gist, a high-level specification language being developed at ISI, has indicated that two of the major impediments to understandability are the unfamiliar

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

syntactic constructs of the language and non-obvious interactions between parts of the specification that are often widely separated. These interactions may cause the specification to denote behaviors that were unintended. This report documents efforts to overcome these impediments by constructing tools to make specifications more understandable, both to specifiers and to those unfamiliar with formal specification languages. One tool, the Gist paraphraser, addresses the syntax problem by directly translating a Gist specification into English. The paraphraser is useful in both clarifying specifications and revealing specification errors. 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. A second tool addresses the more difficult problem of making non-local specification tool interactions apparent by simulating the dynamic behavior implied by the specification. The approach has been to discover non-local interactions by using a symbolic evaluator to analyze a specification. 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.

Table of Contents

List of Figures

1. Introduction

INFORMATION

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[2], has indicated that two of the major impediments to understandability are the unfamiliar syntactic constructs of the language and non-obvious interactions between parts of the specification that are often widely separated. These interactions may cause the specification to denote behaviors that were unintended by the original specifier or not to denote behaviors that *were* intended. This report documents our efforts to overcome these impediments by constructing tools to make specifications more understandable, both to specifiers and to those unfamiliar with formal specification languages.

One tool, the Gist paraphraser, addresses 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 specifiers 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.

The paraphraser deals only with the static aspects of a specification. Our second tool addresses 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 to analyze a specification. The symbolic evaluator gathers and integrates constraints from the different 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 facts directly implied by the specification, but also any further implications that the evaluator may have derived from those 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.

The major observations that have emerged are:

- Good quality English translations of Gist specifications can be achieved without imposing a burden on the specifier. In designing the paraphraser, we recognized that a specifier would have to adhere to certain style restrictions and might have to provide a few annotations to a specification to indicate how it should be translated, but we wanted the style conventions to be as natural as possible and the annotations to be as few as possible. This was done for two reasons: 1) we felt that the paraphraser would be used much more frequently if a specifier could employ it without making extensive modifications to his specification, and 2) if a specification can be translated making only minimal use of annotations, the translation is more likely to accurately reflect the specification. We have found that even specifications written before the creation of the paraphraser can often be translated acceptably (though there is usually room for improvement) because the stylistic conventions imposed by the paraphraser are close to those that specifers follow normally.

- The paraphraser has also proved to be a useful tool for debugging specifications. Originally, we thought the paraphraser would be useful mainly for making specifications understandable to those unfamiliar with Gist. However, we discovered that the paraphraser was also very useful in making specification errors more apparent, even to experienced Gist users. Partly, this is because the English paraphrase is more understandable in most situations, but perhaps more importantly, it gives an alternate view of the specification that makes apparent some aspects of the specification that are not obvious in the formal notation.

- The Gist symbolic evaluator makes some fairly radical departures from the technology that has been developed for symbolic execution of more traditional implementation languages. The main motivation for this was the fact that Gist is based on a predicate calculus view of the world, rather than the implementation view of storage locations containing values. The major departure is the use of a general inference engine to derive all results, rather than a few special functions that simplify expressions or recognize particular cases of impossible paths. A smaller departure is that conditional executions are described in terms of conditional results rather than a large (or infinite) set of separate paths. The main result of these departures is that the Gist symbolic evaluator can discover (and report) many more results which may be of value to the user for the purpose of understanding and debugging, even if they do not directly affect the execution.

  We have found that some of this information is useful in an unexpected way. Often a fact appears which is surprising not because we would have expected it to be false, but because we would have expected a stronger result to be true. Of course, the symbolic evaluator does not say that the stronger result is false, but typically there is a good reason that it cannot be proven, and this points out some interesting (possibly unwanted) feature of the specification.

- Producing English descriptions of symbolic executions is much more difficult than paraphrasing the specification. 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.

    - **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, and conversely, the same referring phrase may be most appropriate for different objects.

The next section presents an example specification and a machine-produced description of its *symbolic evaluation. Chapter 2 describes the tools in detail.* Chapter 3 gives an overview of how the tools are used, and chapter 4 presents an extended example of the use of the interactive symbolic evaluator. This report assumes some familiarity with Gist, although a detailed understanding is not required.

1.1 An Example

The example presented here is a simplified version of a specification for a postal package router (see [6, 10]). The package router is designed to sort packages into bins corresponding to their destinations. A package arrives at a location called the *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 1-1). 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 1-2. It is not necessary to understand the formal notations, since an English translation of the specification (produced by the paraphraser) is available: Figure 1-3 is the English paraphrase of the specification's type structure and Figure 1-4 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 1-5, the user

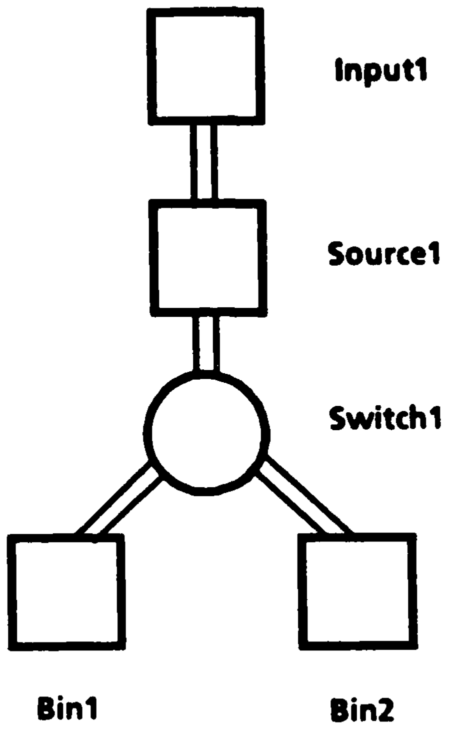


Figure 1-1: Package Router

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 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 1-6).

```
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 1-2: 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 1-3: 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 1-4: English Paraphrase of Possible Actions

```
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 1-5: English Paraphrase of a Test Action

1. A box, call it box1, is inserted.
Result: The new location of box1 is source1.

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.

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 of switch1. Switch1 is not the location of any box.

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.

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.

Figure 1-6: Machine-Produced Description of Symbolic Evaluation of Test

2. Results

An overview of our system is presented in Figure 2-1. To use any of the tools described in this report, a Gist specification must first be translated into the *SAFE representation* using the *Gist to SAFE compiler*. The user may then use the *paraphraser* (described in Section 2.1) to see an English paraphrase of the specification. The *symbolic evaluator* (described in section 2.2) may be invoked to evaluate the entire specification. It produces a history of the execution, called the *trace* which details the results of symbolic evaluation. A user may examine the trace using the *browser* (described in section 2.3) or he may obtain an English summary of the results using the *trace explainer* (described in section 2.4). In addition to the "batch" mode of operation, the symbolic evaluator can also be used in an interactive fashion where the user can direct the evaluation process. This produces a trace tree. Commands are available to the user that allow him to return to previous states and continue the evaluation down different paths. Although most of our work on trace explanation has been directed toward producing explanations of traces produced in the "batch" mode, a limited explanation capability exists for describing the results of each interactive evaluation step. This capability is presented in chapter 4.

2.1 Gist English Paraphraser

This section describes a prototype English paraphraser which can produce English descriptions of program specifications written in Gist. 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 paraphraser 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 paraphraser.

Given a Gist specification and a small amount of additional grammatical information needed for translation (detailed below), the Gist English Paraphraser 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 paraphraser 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 paraphraser can serve as a pedagogical aid by re-

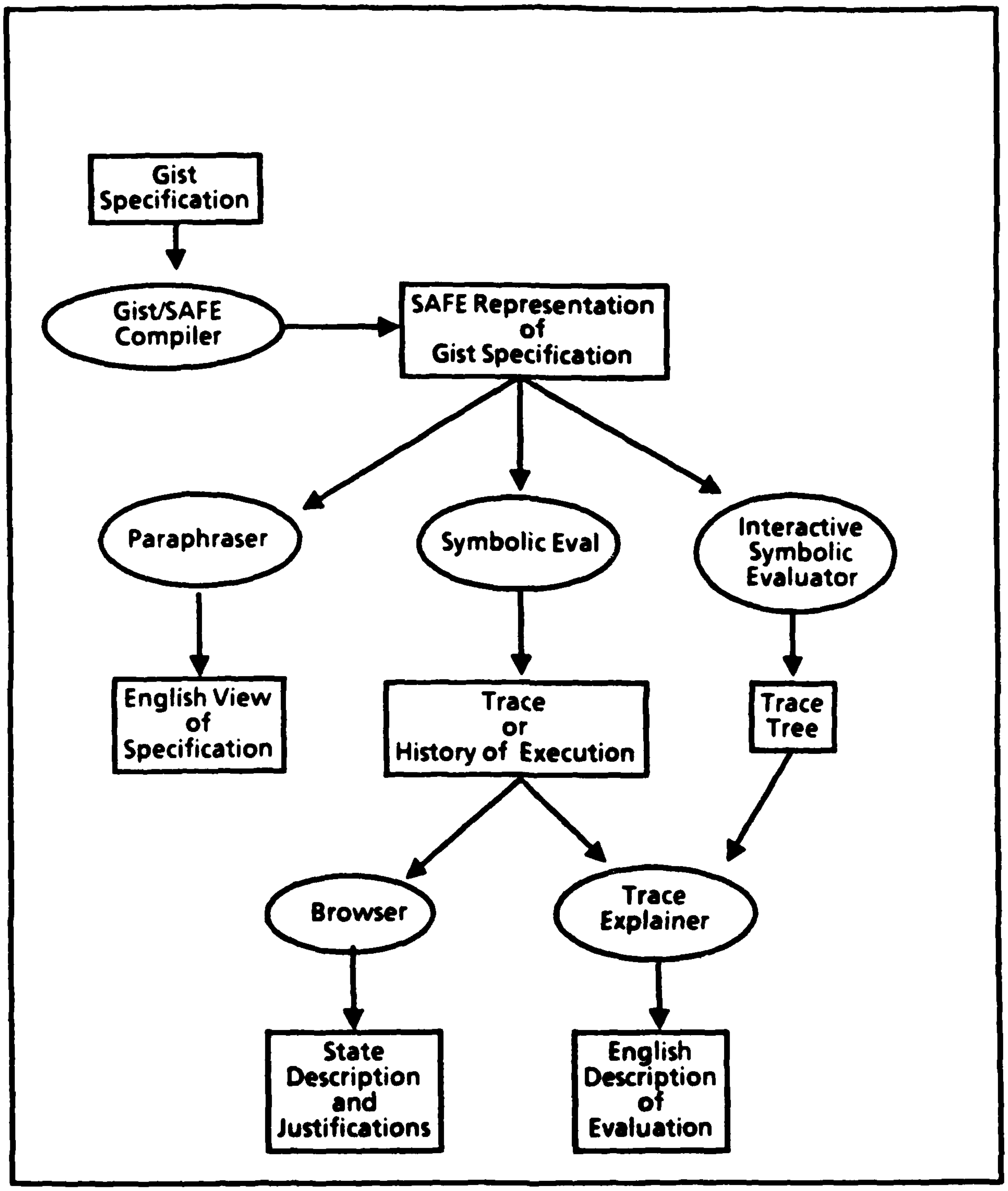

Gist Specification
Gist/SAFE Compiler
SAFE Representation of Gist Specification
Paraphraser
Symbolic Eval
Interactive Symbolic Evaluator
English View of Specification
Trace or History of Execution
Trace Tree
Browser
Trace Explainer
State Description and Justifications
English Description of Evaluation

Figure 2-1: System Overview

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 paraphraser 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. 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 paraphraser 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.

2.1.1 Paraphraser Organization

The Gist English paraphraser uses three passes and an intermediate case grammar representation [4]. The first pass of the paraphraser 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 paraphraser are independent of the particular representation used for Gist specifications. Thus, the portions of the paraphraser that embody English knowledge can be exported to other applications.

Text generation has been investigated by a number of researchers (see [8] for a current bibliography). Boris Katz has produced a generator which is perhaps most similarly structured to the one presented here [7]. 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 paraphraser, 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.

2.1.2 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.

2.1.2.1 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 paraphraser.

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 paraphraser 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 paraphraser 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.

2.1.3 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 paraphraser assumes that it is an English verb. If it is a compound,[1] the paraphraser assumes that the first element of the compound is the verb.

The parameters in an action correspond to cases in a case grammar. The paraphraser knows how to map a fixed set of cases into English. The user must supply the paraphraser 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 paraphraser 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*.)

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

- **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 paraphraser 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."

2.1.4 Examples

This section presents some examples of English produced by the paraphraser. 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 paraphraser 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 in italics.)

(The paraphraser "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 paraphraser 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 paraphraser 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 paraphraser 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 paraphraser split the description up into multiple sentences and introduced the intermediary "p" to make the description clearer.)

2.1.5 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 three topics that we expect will substantially improve the quality of the explanations that can be offered. These are: *global explanation descriptions*, *presentational form*, and *level of abstraction.*

One problem with the current English paraphraser 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 paraphraser 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 paraphraser 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 paraphraser 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 elabcrated 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.

2.2 The Gist Symbolic Evaluator

2.2.1 Introduction

Our approach to symbolic execution regards a specification as a large set of domain axioms, expressed in a first order temporal logic with typed variables. The axioms define the set of acceptable behaviors, i.e., the specified behaviors correspond to the models of the set of axioms. Symbolic execution is a process of forward inference, computing consequences of these axioms. Notice that a specification need not determine the truth or falsehood of every relation, i.e., a relation may be true in some behaviors and false in others.

This approach factors symbolic execution into two processes. First, each statement in the specification is translated into axioms about successive world states. Second, these axioms are used to derive certain interesting consequences, e.g., hidden interactions among different parts of the specification. The success of this approach depends in large part upon the ability of the forward inference engine to find interesting consequences and avoid uninteresting ones. However, the control of forward inference is outside the scope of this paper. The rest of the section describes how various Gist constructs are treated as axioms. We start with primitive constructs and then show how compound constructs are handled in terms of their components.

2.2.2 Constraints

Constraints are the easiest Gist construct to handle, in that they are already in the form of axioms. For example, the constraint that the spouse relation be symmetric is expressed as

$$\forall s_{state}, x_{person}, y_{person} \; (\text{Spouse}(x,y) \supset \text{Spouse}(y,x)) \text{ in } s$$

Actually, in the current implementation, facts about different states are stored separately; more on this later.

2.2.3 Descriptive Reference

Part of the meaning of a Gist statement like the constraint "require Contains(a box, a ball)" is that there must be referents of the object descriptions. Symbolic execution creates a typed "symbolic instance" for each such description. If we call these symbolic instances box1 and ball2, symbolic execution simply proceeds by adding the axiom Contains(box1,ball2). The interpretation in which this makes sense is that box1 and ball2 are not *actually* objects in the world, but rather names of objects. The distinction is that several names can refer to the same object. Thus we *do not* preclude the possibility that ball2 is actually the same object as some other ball that was referred to earlier.

Descriptive reference is merely a constrained form of nondeterministic reference, e.g., requiring a box to contain a red ball is modelled by adding the axioms Contains(box1,ball2) and Red(ball2).

One kind of consequence the symbolic evaluator considers interesting is that two descriptions must (or cannot) refer to the same object. Specifications often contain constraints that imply the identity or non-identity of such descriptions. The most common such constraint requires that a relation be a single-valued function of one of the arguments, e.g., Gender. Another common constraint specifies that a relation describes an optional attribute, e.g., Spouse.

The consequences of uniqueness constraints are found by forward inference. For instance, from Spouse(p1,p2) and Spouse(p1,p3) the system deduces $p2 = p3$. Conversely, from Spouse(p1,p2) and ~Spouse(p1,p3) it deduces $p2 \neq p3$. The system also uses uniqueness constraints to find consequences of facts with universally quantified variables or several arguments to compare, e.g., from Gender(p1,sex1) and Gender(p2,sex2) it deduces $p1 = p2 \supset sex1 = sex2$.

2.2.4 Primitives that Change the World

The most direct effects of primitive changes are easy to axiomatize, e.g., in the state after "insert Spouse(p1,p2)" it is required that Spouse(p1,p2). However, such constraints cannot completely capture the effect of a change. In particular, first order predicate calculus cannot represent the notion that the before and after states are the same except for the effects of the change.

This notion is captured by predicate transformers [3]. The symbolic evaluator stores each state explicitly along with the set of facts known to be true in that state. Facts are propagated between neighboring states. Notice that propagating a constraint backward in time allows the symbolic evaluator to identify its implications for earlier choices. We use Pre(S,F) to denote the consequences our system derives about the state preceding execution of the statement S given that the fact F holds afterward. Similarly, Post(S,F) denotes the consequences about the state resulting from S given that F was true beforehand. For readability, we use the notation "F1 before S => afterwards F2" for "Post(S,F1) = F2" and "F1 after S => beforehand F2" for "Pre(S,F1) = F2".

The computation of pre- and post-conditions is considerably simplified by the following considerations. For any executable statement S and any propositions P and Q:

$\mathrm{Pre}(S,P \wedge Q) \equiv \mathrm{Pre}(S,P) \wedge \mathrm{Pre}(S,Q)$
$\mathrm{Pre}(S,P \vee Q) \equiv \mathrm{Pre}(S,P) \vee \mathrm{Pre}(S,Q)$
$\mathrm{Post}(S,P \wedge Q) \equiv \mathrm{Post}(S,P) \wedge \mathrm{Post}(S,Q)$
$\mathrm{Post}(S,P \vee Q) \equiv \mathrm{Post}(S,P) \vee \mathrm{Post}(S,Q)$

Quantifiers are eliminated by skolemization. This reduces the problem of computing pre- and post-conditions of general propositions to the special case of literals, i.e., positive or negative instances of relations with constants, universally quantified variables, and function applications as arguments. (In the rest of this section, "x" and "y" are universally quantified variables, "f" and "g" are functions, and other unquantified symbols are constants.) The rest of this section describes how this is done for different kinds of primitives.

2.2.5 Changing a Relation

In Gist, the insert and delete statements add and remove relations. The relation is named explicitly, but the objects may be named by description (and thus nondeterministically), e.g., "insert Red(a ball)". Only one instance of a single relation is changed by each such statement; "insert Red(ball1) $\vee$ Green(ball1)" has no meaning. An insertion results in a new state containing the inserted fact. Deletion is treated as insertion of a negated fact.

The problem of computing pre- and post-conditions of other facts with respect to insertion and deletion is simplified by the following considerations. Gist is a first order language, i.e., there are no variables ranging over relations. Thus any literal whose relation differs from the one being inserted or deleted is unaffected. (Gist supports "derived" relations, whose values are changed by changing other relations, but the handling of these is outside the scope of the present paper.) Also, if a positive literal is true before an insertion or a negative literal is true before a deletion, it will still be true afterward:

Red(ball1) before insert Red(ball2) => afterwards Red(ball1)

Finally, if a positive literal is true after a deletion or a negative literal is true after an insertion, it must have been true before:

~Red(ball1) after insert Red(ball2) => beforehand ~Red(ball1)

The only cases in which insertion or deletion changes an existing fact are those in which a literal true beforehand is changed by the insertion or deletion, or a literal that is true afterward is made true by the insertion or deletion. This happens just when the arguments of the fact all refer to the same objects as the corresponding arguments of the relation being changed:

$P(a,x,f(x)))$ before delete $P(b,c,d)$ =>
afterwards $P(a,x,f(x)) \lor (a = b \land x = c \land f(x) = d)$

Recall that all of the variables are universally quantified, so after deleting P(b,c,d), P(a,x,f(x)) is still true for all x with the possible *exception* of c, and that is only an *exception* if $a = b$ and $f(c) = d$.[2]

2.2.6 Changing the Type of an Object

Types in Gist may be thought of as unary relations. Thus "p is a person" corresponds to Person(p). An object's type can be changed by a Gist reclassification statement, e.g., "Pinocchio becomes a person", which corresponds to "insert Person(Pinocchio)".

Quantifiers in Gist always range over objects of a particular type. Therefore a universal statement may have exceptions in neighboring states where more objects have the specified type, e.g.,

$\forall x_{person} \exists y_{person}$ Mother(y,x) before
Pinocchio becomes a person => afterwards
$\forall x_{person} \exists y_{person}$ (Mother(y,x) $\lor$ x = Pinocchio)

This is exactly the effect that arises from treating types as unary relations: $\forall x_{person} \exists y_{person}$Mother(y,x) means
$\forall x \exists y(\text{Person}(x) \supset (\text{Person}(y) \land \text{Mother}(y,x)))$,

where x and y are now untyped variables. The symbolic evaluator would skolemize this to $(\text{Person}(x) \supset (\text{Person}(f(x)) \land \text{Mother}(f(x),x)))$,[3] translate "$\supset$" in terms of "$\lor$" and "~", and apply the rules for computing a post-condition:

$\sim\text{Person}(x) \lor (\text{Person}(f(x)) \land \text{Mother}(f(x),x))$ before
insert Person(Pinocchio) => afterwards

[2]Readers familiar with unification will notice both similarities and differences. For example, P(f(a)) would match with P(f(b)) giving the "substitution" (f(a) = f(b)), i.e., it is not necessary that a = b, just that f(a) = f(b). Also, it is perfectly acceptable to unify x with f(x). In a sense we have a generalized version of unification which can be used for theorem proving, e.g., $P(a,b) \lor Q$ is resolved with $\sim P(c,d) \lor R$ to give $a \neq c \lor b \neq d \lor Q \lor R$.

[3]To make skolemized axioms well-defined, we treat skolem functions as immutable and defined over all objects that ever exist, but do not constrain the value of the skolem function on objects outside the original type.

$\sim$Person(x) $\vee$ x = Pinocchio $\vee$ (Person(f(x)) $\wedge$ Mother(f(x),x))

This is equivalent to the post-condition above.

2.2.7 Creating and Destroying Objects

In Gist, objects are always created with a type. Creating an object is like inserting a type relation except that (1) the created object is different from any object that ever existed before and (2) this object is in only those relations inserted since its creation. (Objects in the initial state may be in arbitrary relationships as long as no constraints are violated.) Similarly, destroying an object is like deleting a type relation except that (1′) the destroyed object differs from any object that ever exists after the destruction, and (2′) destruction deletes all relations in which the destroyed object participated.[4]

These properties of creation and destruction cannot be expressed in first order predicate logic; instead, they are embodied in the inference engine and the predicate transformers. For example, in simplifying an equality the symbolic evaluator checks to see whether one object existed before the other was created. The pre- and post-conditions of creation and destruction combine the effects of reclassification with the requirement that non-extant objects cannot participate in relations.

Notice that the create and destroy statements are almost symmetric in the sense that each viewed backward in time looks like the other. The only difference is that a destroy statement deletes all of the relations involving the destroyed object, whereas the create statement is not empowered to insert arbitrary relations involving the created object.

The following table summarizes the conditions under which creations and destructions invalidate literals. We use P(x,a) as a representative literal, where "a" is a constant and "x" is a universal variable. The occurrences of "a = c" represent the condition that the created or destroyed object (c) is one of the parameters of the relation (we exclude variables since in one case they simplify out and in the other case they are included by the other condition), e.g.,

P(x,a,f(g(x))) before destroy c => afterwards
  P(x,a,f(g(x))) $\vee$ c = a $\vee$ c = f(g(x))

The occurrences of "x = c" represent the condition that the created or destroyed object is (instantiated by) any of the (universal) variables in the literal, e.g.,

P(f(x,y)) before create c => afterward P(f(x,y)) $\vee$ x = c $\vee$ y = c

Additional detail is contained in notes $^{(i - vi)}$ below.

[4] The equality relation can relate non-extant objects and is considered to be immutable. The table below is suitably altered for this case.

Pre- and post-conditions for create and destroy

$P(x,a)$ before create c $\Rightarrow$ afterwards $P(x,a) \vee x = c$ [i]
$P(x,a)$ after destroy c $\Rightarrow$ beforehand $P(x,a) \vee x = c$ [i]
$\sim P(x,a)$ before create c $\Rightarrow$ afterwards $\sim P(x,a)$ [ii,v]
$\sim P(x,a)$ after destroy c $\Rightarrow$ beforehand $\sim P(x,a) \vee a = c \vee x = c$ [v]
$P(x,a)$ before destroy c $\Rightarrow$ afterwards $P(x,a) \vee a = c$ [iii,vi]
$P(x,a)$ after create c $\Rightarrow$ beforehand $P(x,a)$ [iv,vi]
$\sim P(x,a)$ before destroy c $\Rightarrow$ afterwards $\sim P(x,a)$
$\sim P(x,a)$ after create c $\Rightarrow$ beforehand $\sim P(x,a)$

[i] $a = c$ is impossible unless the prior state was devoid of objects of the same type as x, i.e., the quantifier was vacuous. In this case $P(x,a) \vee x = c$ still holds if $a = c$.

[ii] If creations were allowed to insert relations containing the new object, this entry would be "$\sim P(x,a) \vee a = c \vee x = c$".

[iii] $x = c$ reduces to false here since the quantified x only refers to objects that exist.

[iv] Since creation does not insert relations, this case could only arise if insertions were done at the same time as the creation. See section 2.2.10. If creations were allowed to insert relations of the new object, this would be "$P(x,a) \vee a = c$". The $x = c$ reduces to false. See [iii] above.

[v] When P is the equality relation, $x \neq a$, the result is $x \neq a \vee x = c$.

[vi] When P is the equality relation, $x = a$, the result is $x = a$.

2.2.8 Compound Statements

In order to save space we describe only a few problematical constructs. It should be obvious how sequences and procedure calls can be handled. Conditionals are not hard, given a way to represent the truth of the branch condition as of the branching state. It should be mentioned that the branches are combined into a common state after a conditional, i.e., rather than producing a tree of behaviors, the symbolic evaluator describes the state after the conditional in terms of which branch was taken.

2.2.9 Loops

We distinguish between "simple" loops, which can currently be handled and "non-simple" loops which cannot. Simple loops are those in which the iterations are independent of each other, i.e., the same thing is done to each of a set of objects, as in "move all old files off line".

Most loops in implementations are not simple, e.g., "for each file, if age(file)>age(oldestfile) set *oldestfile to file". However, these loops tend not to appear in Gist specifications. They are replaced* by descriptive reference, e.g.,

"a file1 such that $\forall$file2 age(file1)$\geq$age(file2)"

Simple loops are symbolically executed for the entire set at once. Basically, the loop variables turn into universally quantified variables in the facts that are inserted. After "if file1 is old move it offline" we know old(file1)$\supset$offline(file1), whereas after "for all files F, if F is old move it offline" we know $\forall x_{file}$ old(x)$\supset$offline(x). All of the symbolic instances that are generated in a loop are skolem functions of the loop variables. In general the computation of pre- and post-conditions introduces existential quantifiers, but is otherwise similar to the versions described above, e.g.,

$\sim P(a,b)$ before insert $P(c,d)$ => afterwards
  $\sim P(a,b) \vee (c = a \wedge d = b)$,   whereas
$\sim P(a,b)$ before insert $P(x,f(x))$ => afterwards
  $\sim P(a,b) \vee (\exists x\ x = a \wedge f(x) = b))$

2.2.10 Atomic Statements

The Gist "atomic" construct combines the effects of several constituent statements into a single state transition. An example is the marriage action that simultaneously inserts two spouse relations. It would not have been sufficient to insert one at a time because this would have led to an intermediate state of the world that violated the constraint that the spouse relation be symmetric. (Actually, that specification would still have been consistent, but now it would be possible to marry two people only if they were already each others' spouse -- another interesting result of symbolic execution.) Of course, the constituent statements of an atomic must themselves cause no more than one state transition.

The facts that become true because of the statements in the atomic must all be true in the final state. e.g., if an atomic contains both insert P(a) and delete P(b), then a and b must be distinct. This points out a difference between executing two statements atomically and executing them in either order. There is no problem with inserting P(a) and then deleting it. A fact that is propagated through an atomic can be affected by any combination of the statements in the atomic. The pre- or post-condition of a fact with respect to an atomic statement is the disjunction of the pre- or post-conditions of the fact with respect to each constituent statement.

2.2.11 Summary

We have described a system that characterizes the behaviors permitted by a formal specification containing such constructs as descriptive reference, nondeterminism, and constraints. It translates a specification into a set of axioms and uses forward inference to compute interesting consequences of them. It uses predicate transformers to propagate facts between neighboring states; the computation of pre- and post-conditions in the relational database model has, to the author's knowledge, never been described before.

We have been pleasantly surprised to find that, although many problems that arise are very difficult (or even impossible) to solve in general, the most common and useful cases tend to be the easiest. We have also found that a high level specification can be easier to execute symbolically than a low level program. In retrospect this is not surprising, since the characterization of low level implementations involves a lot of work that could be described as de-compilation.

The decision to represent each state explicitly imposes certain limitations. In particular, arbitrarily long sequences of states cannot be represented. This precludes the description of non-simple loops and certain types of historical reference. Historical reference (a special case of descriptive reference) is not yet handled. We also currently do not attempt to handle the arbitrary interleaving and merging of lines of control provided by Gist. We hope to attack these problems, but a great deal can be done without solving them.

2.3 The Trace Browser

This section describes the Trace Browser. An example of its use is presented in the appendix. The Browser is designed to allow a user to examine the trace generated by the symbolic evaluator. The user can move around in the trace. examine the state at different points in the evaluation, and obtain justifications for facts in states. Since the total state of evaluation can become quite large, the Browser also allows the user to indicate particular objects he is interested in, and display only the information applying to them.

The browser is a relatively low-level tool that presents information about the trace in a form close to its internal representation. For that reason, it is primarily useful for users willing to invest the time required to gain a detailed understanding of the symbolic evaluator and its representations. Most users will probably prefer the Behavior Explainer (described in the next section) which produces natural language translations of the internal trace representation.

The browser's commands for moving about the trace work at the level of Gist statements, although the actual trace is considerably finer-grained. The browser works at this coarser level so that there will be a close correspondence between what the moves a user can make and his specification.

Commands for the browser are:

- *In* move into a compound statement (such as Begin or If).

- *Out* the opposite of In.

- *Next* moves to the next statement in a sequence of statements. Remains at the same level of embedding. That is, this command will not move beyond the last statement in a Begin Block.

- *Previous* the oppositie of Next.

- *Run Step* similar to Next, but if invoked at the end of a block, it will automatically perform Outs until an executable step is found, then does a Next.

- *Top* returns to outermost statement

- *Show State* shows the state as of the current position. Asks the user whether the state should be shown before or after execution of the current statement, and whether final or incremental knowledge should be displayed. If the user requests final knowledge, the display may include facts propagated back from states yet to come. Incremental will display only the knowledge known to the Evaluator when it first entered the current state.

- *Examine Single Object* prompts user for the object to be displayed, and shows its attributes, how it is used, and relations in participates in.

- *Add Object to Monitored Objects* This command allows the user to indicate to the browser which objects he is interested in.

- *UnMonitor Object* removes an object from the list of monitored objects.

- *Display Monitored Objects* does what the name implies.

- *Justify Fact* displays a justification for a fact, that is, the reasoning the Evaluator went through in determining that a particular fact was true. Often, a fact may be justified by other facts. In that case, the browser will display them, and the user may ask to see justifications for those facts, too.

2.4 The Gist Behavior Explainer

Above, we described the Gist paraphraser. This addresses 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. But the paraphraser deals only with the static aspects of a specification. This section deals with the more difficult problem of making non-local specification interactions apparent by simulating the dynamic behavior implied by the specification.

As the symbolic evaluator evaluates a specification, the trace it creates describes the dynamic interactions we want to be able to describe to users. 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. As stated in the introduction, these problems include:

- **Detail suppression.**

- **Proof summarization and reformulation.**

- **Referring expressions.**

Figure 2-2 reproduces (with annotations) the example presented in Section 1. This example will be used to illustrate the initial solutions we have found for the problems listed above.

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 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.

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

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 has 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 2-2, continued

2.4.1 System Organization

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 [13]).

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.

2.4.2 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.

2.4.2.1 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.

2.4.2.2 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 [12]: 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.

2.4.2.3 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*.

2.4.3 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 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 reasoning 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 [9], 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.

3. Detailed System Structure

This chapter presents a more detailed view of the system structure, and outlines instructions for using the symbolic evaluator, paraphraser and trace explainer.

3.1 Composition of the Symbolic Evaluator

The Gist symbolic evaluator makes use of several large (and complex) systems which are described only briefly here for the sake of context.

Interlisp [11] is the programming language and environment in which all of the programs discussed below are written. The Gist language is parsed by the POPART system [14], which also provides a Gist editor. The "Gist to Safe compiler" reads the parse tree and produces an intermediate file (a "gcom" file) which can be loaded into the symbolic evaluator. The AP3 system [5] provides a relational database, a type hierarchy, demon triggering and contexts. It is used by the Hearsay3 system [1], which is a tool for building "expert systems". A "SAFE system" (the name is of historic origin and unrelated to safety) is obtained by adding programs for reading gcom files and translating them into an AP3 database, called the "SAFE representation" of the specification. The symbolic evaluator works from this representation of the original Gist specification and is built from a SAFE system.

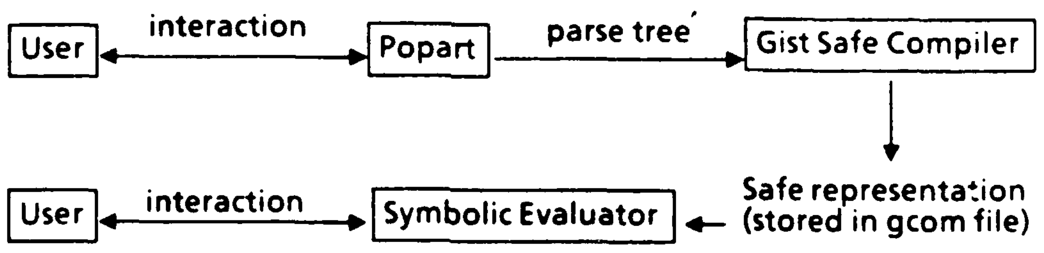


Figure 3-1: Symbolic Evaluator Organization

The specification testing tool that was built for the present contract is the bottom part of Figure 3-1.

The symbolic evaluator itself is made up of several major parts. The English descriptions of the specification and behavior are generated from a case grammar by programs in the file "case". The file "ENGCON" contains the programs that express information in the case grammar. For example, there are programs to translate various Gist constructs and programs to translate the facts found in symolic execution. The file "ETREE" contains programs that maintain the execution tree and accept user instructions for altering it, moving around in it and displaying it. The file "LOGIC" contains the theorem prover that deduces new facts within a state. The file "MTRACE" contains the programs that maintain the internal representation of the execution history (called "the trace") and compute pre- and post-conditions. The file "MEVAL" contains the programs that understand how to "execute" the SAFE representation of a specification in terms of building an execution history. The file "TBROWSE" provides an interface through which the trace explainer (in "ENGCON") can examine the execution history (recorded by "MTRACE"). These relationships are outlined in Figure 3-2.

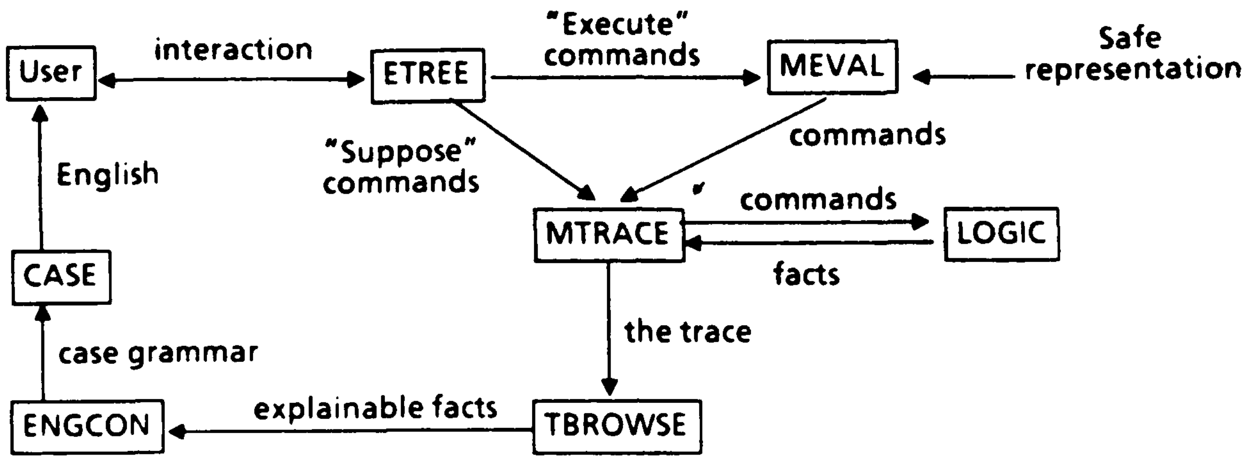


Figure 3-2: System File Structure

3.2 Use of the Symbolic Evaluator

The symbolic evaluation system is meant as a tool for helping a user to understand a Gist specification. The meaning of a Gist specification is the set of behaviors that it denotes. (A behavior is defined as a sequence of world states, where a world state is defined by the set of objects that exist and the relations among those objects. The symbolic evaluator helps a user to explore that set of behaviors. The current version does not allow concurrency as defined by the semantics of Gist. It only handles behaviors that consist of a sequence of action and demon invocations. Furthermore, the current system does not attempt to enumerate all such sequences, but forces the user to decide which ones he wants to examine.

The result of symbolically executing a sequence of action and demon invocations is a description of the set of **all** *behaviors containing that sequence of invocations with no other activities intervening. It is possible to include constraints with the actions and demons, in which case the result is a description of the subset of those behaviors that satisfy the constraints.*

3.2.1 The Model of the Exploration Process

Gist specifications tend to contain a number of "global constraints", facts that are required to be true of every state in any allowable behavior. The first step in understanding a specification is to understand the global constraints and their consequences (which are also global constraints by the definition above). The global constraints define the set of all states that could be part of any allowable behavior. This is an appropriate set of states from which to start the exploration.

The set of behaviors is explored by building an "execution tree", starting from the most general possible state. Every node in the tree represents a set of behaviors which has followed some sequence of events, subject to some set of constraints. From any such node, it is possible to generate a more specialized node, consisting of a subset of the behaviors of the original node. These specializations can take two forms. The first is a constraint. Given a set of behaviors, adding a constraint (in the form of a statement in first order logic) generates the subset that satisfy the constraint at the last state. The other type of specialization is the addition of an event. This generates the subset of the original set of behaviors that is followed by the added event. (Note that we consider our descriptions of sequences of states to also describe, incompletely of course, longer sequences that contain the described sequence.)

3.2.2 Commands

The most important command is the Help command. It allows you to find and find out about all the other commands. Just typing "Help" will tell you what words the Help command can help you with. Typing Help followed by one of those words will tell you about that word, e.g., if it's a command, then what does it do and how can you use it. The words that Help knows about include the following commands.

- GetGlobalFacts processes the global constraints and describes any interesting consequences of them. It also initializes the execution tree with the null execution.

- ShowTree prints a representation of the execution tree.

- ShowPath prints a representation of the path from the root to the "current execution", the node in the tree where you are now.

- MoveTo allows you to move to another node in the execution tree.

- Suppose allows you to add a constraint to the current execution. It describes any interesting consequences of the constraint and leaves you at the node after the constraint. (Details of how to type in constraints are described under the Help for ReadWff.)

- Execute allows you to add the execution of a demon or action to the end of the current execution. Actually, this will typically consist of many events. The symbolic evaluator will keep you up to date on where it is executing. Every so often it will ask whether it should keep going or stop. It would be too much trouble to ask at every event. However, you can always MoveTo one of the events that it passed, add a constraint at that point and then continue.

- Continue allows you to continue an execution that is not complete.

3.3 Using the Paraphraser and Trace Explainer

As mentioned in chapter 2, the user must supply a few annotations to achieve good translations with the paraphraser. There are two kinds of annotations: *translation annotations* that indicate how a particular Gist construct should be translated and *grammatical annotations* that indicate irregular verb forms, spellings, and so forth.

The trace explainer uses the same files and one additional one: T-TRACE. The annotations required for the paraphraser are also sufficient for the trace explainer. The trace explainer is invoked by the the function DESCRIBE-TRACE when the system is used in batch mode, and the explainer is automatically invoked in interactive mode.

3.3.1 Translation Annotations

The primary annotations the user must supply are the case annotations for actions that indicate what cases the parameters of actions should take. The currently defined cases are:

- **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*.)

The record *ACTCASES* has been defined using the Interlisp record facility to aid the user in supplying cases for action parameters. To indicate the cases for an action, the user replaces the ACTCASES record of the action with a list, where each element of the list is the case to be used in translating the corresponding parameter of the action. For example, to annotate an action like:

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

the user would say: **(replace ACTCASES of un:MoveShip with '(OBJECT DIRECTIONAL))**

The LOCATIVE case requires that a preposition be supplied. This is using a dotted pair, where the CAR is LOCATIVE and the CDR is the preposition.

To indicate that an attribute should be translated as a verb, the atom corresponding to the lower-case attribute name is given the VERBPROP property. Thus, in the port manager specification, the paraphaser is told to translate the attribute "handle" as a verb by saying:

```
     (PUTPROPS handle VERBPROP T)
```

3.3.2 Grammatical Annotations

The English generator has a limited automatic morphological component, but for irregular spellings and verb forms the user must supply annotations. These notations are placed on property list of the lower case singular form of nouns and the infinitive form of verbs. If the verb or noun is regular, no notation need be supplied. Current notations are:

- **3SINGULAR** The third person singular form of a verb.
- **SPAST** The singular past form of a verb.
- **PPAST** The plural past form of a verb.
- **GERUND** The gerund form of a verb.
- **PLURAL** The plural form of a noun.

4. Using the Interactive Evaluator: an Example

This appendix shows some transcripts of sessions with the symbolic evaluation system. Commentary is inserted in italic type.

We begin with a very small specification. The Help command is used extensively here to introduce the available commands. For reference we include the Gist specification:

```
begin
 type sex()
  definition{Male, Female};
 type person(gender | sex, spouse | person :optional ::optional);
 always prohibited Spouses←of←same←sex there exists person ||
                     person :gender = person :spouse :gender;
 always required Spouse←is←Commutative for all x | person,
                          y | person ||
                     x :spouse = y => y :spouse = x;
 agent bookkeeper()
  where action Marry[p1 | person, p2 | person]
      definition atomic insert p1 :spouse = p2;
                insert p2 :spouse = p1
           end atomic
  end
end
```

We now proceed to the terminal session. The user inputs are numbered by the underlying Lisp system and extend to the end of the line.

```
24←Help
Documentation exists for the following:
(Continue Execute Finalize General GetGlobalFacts Help MoveTo ReadWff
     ShowCurrentState ShowFinalFacts ShowNames ShowPath ShowSpec ShowTree
     Suppose)
25←Help General
(General NIL NIL)

This symbolic execution system allows you to build and examine a tree of
executions. First you must LOAD the "gcom" files containing the specification.
Then do (GetGlobalFacts) to create the root node (named ExecutionTree) in
which the global constraints have been processed but nothing has happened.
At any point thereafter you may examine the current execution, continue the
current execution or start a new one, add new contraints to the current
execution or move to another execution in the tree. The Help command can
provide more information.
NIL
26←LOAD(gcom/marry)

expanding LISTP, 65520 used, 2228224 before GC
```

These are storage allocation messages. The GCGAG below turns them off. They are turned off in the other examples.

```
expanding LISTP, 131032 used, 2162688 before GC
/remote/don/gcom/marry
27←GCGAG()
T
28←Help ShowSpec
(ShowSpec NIL NIL)

(ShowSpec) prints an English translation of the currently loaded specification.
In the current system it is a bad idea to load more than one specification into
the same system.
NIL
29←ShowSpec()
 There are sexes, persons and bookkeepers.

 Male and female are the only sexes.

 Each person has one gender which is a sex. Each person may have a spouse
 which is a person. Each person is optionally the spouse of a person.

 A bookkeeper can marry.
```

Bookkeeper is the type of agent which can perform marriages. In this specification, bookkeepers are not people and can not be married but can only perform the marry action. In the next example, the English generator will be instructed how to describe actions other than simply naming them.

```
   To marry:

      Action:
      Do atomically:
      1. Add: The p2 becomes the p1's spouse.
      2. Add: The p1 becomes the p2's spouse.
```

Here again, better English could have been generated by instructing the English generator (or picking better names in the specification). The parameters of the action, p1 and p2, are identified as parameters by the article "the".

```
Constraint (Spouse is Commutative):

   Always required:

      For all persons y and persons x:

          If: The spouse of person x is person y, Then: The spouse of
          person y must be person x.

Constraint (Spouses of same sex):
   Always prohibited:

      There exists a person:

         Where:

            The person's gender is the gender of the person's spouse.

NIL
30←Help GetGlobalFacts
(GetGlobalFacts NIL NIL)

(GetGlobalFacts) initializes the execution tree by creating a root node in
which all the global constraints are collected, but nothing has been executed.
NIL
31←GetGlobalFacts()
(processing global constraint Spouse←is←Commutative)
(processing global constraint Spouses←of←same←sex)
(processing nonzero countspec for gender)
(processing uniqueness countspec for gender)
(processing uniqueness countspec for spouse)
(processing uniqueness countspec for spouse)
(processing definition of type sex)
(News about previous state)
Result: For all persons, p1, the spouse of p1 is not p1.
NIL
```

In general, the symbolic evaluator tries to tell the user what it's doing with the messages in parentheses. In general, any step in symbolic execution may result in new information about the state from which the step started and about a new state that was generated. Information about the state from which the step started is printed as "News about previous state" and information about the new state prints as "News about current state." In this case, no new state was generated, but it did find out something about the initial state. The current system does not attempt to justify its conclusions to the user.

```
32←Help ShowPath
(ShowPath NIL NIL)

(ShowPath ExecutionNode)  [LAMBDA SPREAD]
ShowPath generates a brief description of the path from the root to the
argument node. (ShowPath) is the same as (ShowPath CurrentExecution).
Nodes are named Ex # 1, Ex # 2, etc. These are atoms whose values are suitable
arguments for ShowPath, ShowTree, MoveTo, etc. The nodes from which it is
possible to Continue are shown in parentheses, and anomolous nodes are shown
as (Ex # 3 Anomolous).
NIL
33←ShowPath()
(Initialize Ex # 1)
```

We now proceed to a somewhat larger specification, demonstrating such features as the instruction to the English generator and the execution of an action. Again we start with the Gist version.

```
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 Move[ship, pier]
      precondition ship ::Harbor = manager :Port
      precondition manager :Port :Pier = pier
      definition update :Berth of ship to pier :Slip;
     action Load[ship, cargo]
      precondition ship :Berth ::Slip :Handle = cargo
      precondition ship ::Harbor = manager :Port
      definition insert ship :Carry = cargo       ;
     action Assign[cargo, port]
      definition Load[port ::Destination, cargo]
  end
end                                           ..
```

Here is the terminal session.

```
33←LOAD(gcom/ships)
/remote/don/gcom/ships
34←LOAD(shipinit)
File Created:30-JUN-83 12:00:40
SHIPINITCOMS
/remote/don/shipinit
35←SHIPINITCOMS
((P (replace ACTCASES of  un:Move with '(OBJECT DIRECTIONAL)))
 (P (replace ACTCASES of  un:Load with '((LOCATIVE . ON)
                          OBJECT)))
 (P (replace ACTCASES of  un:Assign with '(OBJECT DIRECTIONAL)))
 (PROP VERBPROP carry handle harbor)
 (PROP 3SINGULAR carry))
```

This shows what instruction the English generator was given:

- *It was told how to use the parameters in describing the actions.*

- It was told to use the the attributes carry, handle and harbor as verbs.

- It was given the third person singular form of carry (carries).

36←ShowSpec()
There are ports, piers, slips, 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.

Each pier handles multiple cargos. Each pier has multiple slips. Each slip belongs to one pier.

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

Fuels and grains are cargos.

Each manager has one port. Each port belongs to one manager. A manager can move a ship, load a cargo or assign a cargo.

To move a ship to a pier:

Action: The ship's berth is updated to a slip of the pier.

Preconditions:

The pier must be a pier of the manager's port.
The manager's port must harbor the ship.

To load a cargo on a ship:

Action: Add: The ship carries the cargo.

Preconditions:

The manager's port must harbor the ship.
The pier that has the ship's berth must handle the cargo.

To assign a cargo to a port:

Action: Load the cargo on a ship whose destination is the port.

Constraint (Fuel And Grain):

Always prohibited:

There exists a fuel f, a grain g and a ship s:

```
         Where:

            Ship s carries grain g and fuel f.

 Constraint (Berths Are In Ports):

   Always required:

      For all ships s:

         If: Ship s has any berth,
         Then: A port p harbors ship s.  The berth of ship s must be a
         slip of a pier of p.

NIL
37←GetGlobalFacts()
(processing global constraint Fuel←And←Grain)
(processing global constraint Berths←Are←In←Ports)
(processing nonzero countspec for Port)
(processing uniqueness countspec for Port)
(processing nonzero countspec for Port)
(processing uniqueness countspec for Port)
(processing nonzero countspec for Destination)
(processing uniqueness countspec for Destination)
(processing uniqueness countspec for Berth)
(processing uniqueness countspec for Berth)
(processing nonzero countspec for Handle)
(processing nonzero countspec for Slip)
(processing uniqueness countspec for Slip)
(processing nonzero countspec for Slip)
(processing nonzero countspec for Pier)
(processing uniqueness countspec for Pier)
(processing nonzero countspec for Pier)
(processing uniqueness countspec for Harbor)
NIL
38←Execute(Load)
 To load a cargo on a ship:

   Action: Add: The ship carries the cargo.

   Preconditions:

      The manager's port must harbor the ship.
      The pier that has the ship's berth must handle the cargo.

The agent is some manager, call it manager1.  The ship is ship1.  The cargo
is cargo1.
(processing precondition)
The manager's port harbors the ship.
```

```
(processing precondition)
 The pier that has the ship's berth handles the cargo.
(News about previous state)

Result: The berth of ship1, call it slip1, is a slip of some pier, call it
 pier1.  Pier1 handles cargo1.  The port of manager1 is some port, call it
 port1.  Port1 harbors ship1.

Continue with: Add: The ship carries the cargo.
(News about current state)

Result: Ship1 carries cargo1.

If: Cargo1 is any grain
   Ship1 does not carry any fuel,

If: Cargo1 is any fuel
   Ship1 does not carry any grain,

 Cargo1 can not be a grain and a fuel.
NIL
39←ShowPath()
(Initialize Ex#1 (Execute  un:Load NIL)
        (Ex#2)
        (Continue  un:INSERT-77)
        Ex#3)
```

Notice that there are three execution contexts, one before the Load, one after the preconditions have been processed and one after the load is done. The general goal was to provide an execution context for every point that a user might want to examine or move to.

```
40←ShowNames()
((Actions Move Load Assign)
 (Demons)
 (Agents manager)
 (Types fuel grain cargo ship slip pier port State CHARACTER INTEGER ENTITY)
 (Relations Port Carry Destination Berth Handle Slip Pier Harbor Branch RANDOM)
 (Literals)
 (GeneratedObjects (port1 of type port)
            (pier1 of type pier)
            (slip1 of type slip)
            (cargo1 of type cargo)
            (ship1 of type ship)
            (manager1 of type manager)))
```

At this point the user has access to such names as ship1. These can be used either for adding constraints with the Suppose command, or for passing parameters to new actions in the Execute command.

The next example features more execution, movement in the execution tree and some more complexities in symbolic execution. Again, we start with the Gist version.

```
begin
 type box(Location | location, Destination | bin);
 type location()unique
        supertype of<input()
               definition{input1};
               source(SourceOutlet | switch)
               definition{source1}       ;
               internal←location()unique
                         supertype of<switch(Setting |


                         internal←location        ,
                         Outlet |
                         internal←location :multiple)
                         ;
                         bin()>
        >
 ;
 always prohibited multiple←boxes←at←source there exists box.1, box.2 ||
                        box.1 :Location = source1
                        and box.2 :Location = source1 ;
 always required switches←set←to←outlets for all s | switch ||
                       s :Setting = s :Outlet ;
 agent PackageRouter()
  where action Insert[box]
      definition update :Location of box from input1 to source1;
     action Set[switch]
      precondition~$ :Location = switch
      definition update :Setting 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 :SourceOutlet
             else update :Location of box to box :Location :Setting
  end
end
..
```

Here's the terminal session.

```
25←LOAD(gcom/prtspec2)
/remote/don/gcom/prspec2
26←ShowSpec()
There are boxes, locations and packagerouters.

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

Internal←locations, sources and inputs are locations.
```

Bins and switchs are internal←locations.

Each switch has one setting which is an internal←location. Each switch has multiple outlets which are internal←locations.

Source1 is the only source. The source has one sourceoutlet which is a switch.

Input1 is the only input.

A packagerouter can insert, set or move.

To insert:

Action: The box's location is updated from input1 to source1.

To set:

Action: The switch's setting is updated to an outlet of the switch.

Preconditions:

The switch must not be the location of any box.

To move:

Action:

If: The box's location is source1,
Then: The box's location is updated to the sourceoutlet of source1.
Else: The box's location is updated to the setting 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.

Constraint (switches set to outlets):

Always required:

For all switchs s:

The setting of switch s must be an outlet of switch s.

Constraint (multiple boxes at source):

```
    Always prohibited:

        There exists a box.2 and a box.1:

            Where:

                1. The location of box.1 is source1, and
                2. The location of box.2 is source1.

NIL
27←GetGlobalFacts()
(processing global constraint switches←set←to←outlets)
(processing global constraint multiple←boxes←at←source)
(processing nonzero countspec for SourceOutlet)
(processing uniqueness countspec for SourceOutlet)
(processing nonzero countspec for Setting)
(processing uniqueness countspec for Setting)
(processing nonzero countspec for Outlet)
(processing nonzero countspec for Location)
(processing uniqueness countspec for Location)
(processing nonzero countspec for Destination)
(processing uniqueness countspec for Destination)
(processing definition of type source)
(processing definition of type input)
NIL
28←ShowNames()
((Actions Insert Set Move)
 (Demons)
 (Agents PackageRouter)
 (Types bin switch internal←location source input location box State CHARACTER
     INTEGER ENTITY)
 (Relations SourceOutlet Setting Outlet Location Destination Branch RANDOM)
 (Literals (input1 of type input)
        (source1 of type source))
 (GeneratedObjects))
29←Suppose((A (b box)(Location b input1)))
```

At the moment we can only pose constraints in the form of predicate calculus. This constraint would be literally translated as "for all b of type box, the location of b is input1".

```
(News about previous state)
Result: The location of every box is input1.
NIL
30←Suppose((E (s1 switch)(A (s2 switch)(= s1 s2))))
```

Again, the literal translation is "there is a switch s1 such that for every switch s2, s1 and s2 are the same object". In other words, there is exactly one switch. The explainer makes up the name switch1 for this switch.

```
(News about previous state)
```

```
Result: Switch1 is the only switch.
NIL
31←Execute(Insert)


 To insert:

    Action: The box's location is updated from input1 to source1.

 The agent is some packagerouter, call it PackageRouter1.  The box is box1.

Continue with: The box's location is updated from input1 to source1.
(News about current state)

Result: The new location of box1 is source1.
NIL
32←Execute(Insert)


 To insert:

    Action: The box's location is updated from input1 to source1.

 The agent is some packagerouter, call it PackageRouter2.  The box is box2.

Continue with: The box's location is updated from input1 to source1.
(Anomoly Encountered)
```

Again, the explainer does not currently justify the results to the user. The anomoly encountered is that two inserts in a row would result in two boxes being at the source, which violates a constraint.

```
NIL
33←ShowPath()
(Initialize Ex # 1 (Suppose (A (boxx99)
                   (Location boxx99 input1)))
      Ex # 2
      (Suppose (E (switchx100)
              (A (switchx101)
                (= switchx100 switchx101))))
      Ex # 3
      (Execute  un:Insert NIL)
      (Ex # 4)
      (Continue  un:UPDATE-67)
      Ex # 5
      (Execute  un:Insert NIL)
      (Ex # 6)
      (Continue  un:UPDATE-67)
      (Ex # 7 Anomolous))
34←MoveTo(Ex # 5)
```

NIL

We see that from Ex # 3 the first Insert ended up at Ex # 5, and from there another insert turned out to be anomolous. We return to the situation after the first insert with the MoveTo command.

```
35←Execute(Move)


 To move:

   Action:

   If: The box's location is source1,
      Then: The box's location is updated to the sourceoutlet of source1.
      Else: The box's location is updated to the setting 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.

 The agent is some packagerouter, call it PackageRouter3. The box is box3.
(processing precondition)

 Either:
  1. The box's location is source1, or
  2. The box's location is a switch.
(News about previous state)

Result: Box3 is box1.

Continue with:

 If: The box's location is source1,
   Then: The box's location is updated to the sourceoutlet of source1.
   Else: The box's location is updated to the setting of the switch that
   is the box's location.
(suppose branch condition is true)
 The box's location is source1.

Continue with: The box's location is updated to the sourceoutlet of source1.
(News about current state)

Result:

If: The location of box1 was source1,

   Then: The new location of box1 is switch1.
```

```
(leaving conditional area)
(suppose branch condition is false - NOT the case that:)
 The box's location is source1.
(This Branch is Impossible)
(leaving conditional area)
(News about current state)

Result: The new location of box1 is switch1.
NIL
36←ShowPath()
(Initialize Ex # 1 (Suppose (A (boxx99)
                   (Location boxx99 input1)))
       Ex # 2
       (Suppose (E (switchx100)
             (A (switchx101)
               (= switchx100 switchx101))))
       Ex # 3
       (Execute  un:Insert NIL)
       (Ex # 4)
       (Continue  un:UPDATE-67)
       Ex # 5
       (Execute  un:Move NIL)
       (Ex # 8)
       (Continue  un:CONDITIONAL-55)
       (Ex # 9)
       (Continue  un:UPDATE-47)
       (Ex # 10)
       (NextStatement  un:UPDATE-47)
       (Ex # 11)
       (Continue  un:UPDATE-54)
       (Ex # 12)
       (NextStatement  un:UPDATE-54)
       Ex # 13)
```

Notice that there are actually a lot of execution contexts here. Some are inside an impossible branch--no state information was printed for those. In addition to contexts related by Continuation which we saw earlier, there are contexts related by "NextStatement". Ex # 8 is the context in which the preconditions of Move have been processed. Ex # 9 is the context in which the branch condition is assumed true, Ex # 10 is the context in which the branch condition is assumed to have been true and the update has been done. Ex # 11 is the context in which the branch condition is assumed false. Ex # 12 is the context in which the branch condition is assumed false and the other update has been done. Ex # 13 is the context in which the entire conditional (and the entire Move) has been done.

```
37←ShowTree()
(Initialize Ex # 1 ((Suppose (A (boxx99)
                   (Location boxx99 input1)))
       Ex # 2
       ((Suppose (E (switchx100)
             (A (switchx101)
        (= switchx100 switchx101))))
Ex # 3
((Execute un:Insert NIL)
 (Ex # 4)
 ((Continue un:UPDATE-67)
  Ex # 5
  ((Execute un:Move NIL)
   (Ex # 8)
   ((Continue un:CONDITIONAL-55)
    (Ex # 9)
    ((Continue un:UPDATE-47)
     (Ex # 10)
     ((NextStatement un:UPDATE-47)
      (Ex # 11)
      ((Continue un:UPDATE-54)
       (Ex # 12)
       ((NextStatement un:UPDATE-54)
        Ex # 13 NIL)))))
  (Execute un:Insert NIL)
  (Ex # 6)
  ((Continue un:UPDATE-67)
   (Ex # 7 Anomolous)
   NIL)))))))
```

5. Appendix

The appendix shows how the trace browser can be used. As was mentioned in the text, the trace browser is a very low level tool that will be most useful to those willing to invest the time required to become familiar with the low level representations used by the symbolic evaluator.

```
88←BR()

 un:Test

 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 sourceoutlet 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.
```

The browser is started, and prints out the action to be executed.

```
>show State before or after execution of current statement?before
Final or Incremental knowledge?incremental
```

The user asks to see the incremental state before the current statement was executed. The incremental state contains just those facts that could be found to be true when this statement was first encountered. That is, the incremental state does not contain any facts that have to be true due to statements that will be encountered in the future.

```
1     (OR (= bin2 (bin100 boxx12)) (= bin1 (bin100 boxx12)))
```

```
2     (SourceOutlet sourcex32 switch1)

3     (= switch1 (switch104 sourcex32))

4     (Setting switchx27 (internal←location103 switchx27))

5     (Outlet switchx22 (internal←location102 switchx22))

6     (Location boxx17 (location101 boxx17))

7     (Destination boxx12 (bin100 boxx12))

DisplayState

>show State before or after execution of current statement?before
Final or Incremental knowledge?final
```

The user asks to see the "final" knowledge about the state before execution of the current state. Final knowledge is like incremental, but it includes facts that the system will later discover have to be true now. In this example, facts 1, 2 and 3 are all true due to preconditions in the test action that will be evaluated shortly.

```
1     (Outlet switch1 bin1)

2     (Outlet switch1 bin2)

3     (Location boxx38 input1)

4     (OR (= bin2 (bin100 boxx12)) (= bin1 (bin100 boxx12)))

5     (SourceOutlet sourcex32 switch1)

6     (= switch1 (switch104 sourcex32))

7     (Setting switchx27 (internal←location103 switchx27))

8     (Outlet switchx22 (internal←location102 switchx22))

9     (Location boxx17 (location101 boxx17))

10     (Destination boxx12 (bin100 boxx12))


>in
```

"In" moves the browser inside the action so it can be explored step by step.

```
un:LOGICALQUANT-40
For all boxes:

    The box's location is input1.


>next
```

"Next" moves to the next step.

```
 un:RELATIONSHIP-37
 The sourceoutlet of source1 is switch1.

>next

 un:RELATIONSHIP-36
 Bin2 is an outlet of switch1.

>next

 un:RELATIONSHIP-35
 Bin1 is an outlet of switch1.

>next

 un:BLOCK-41


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.

>show State before or after execution of current statement?before
Final or Incremental knowledge?incremental

1     (Outlet switch1 bin1)

2     (Outlet switch1 bin2)

3     (Location boxx38 input1)

4     (OR (= bin2 (bin100 boxx12)) (= bin1 (bin100 boxx12)))

5     (SourceOutlet sourcex32 switch1)

6     (= switch1 (switch104 sourcex32))

7     (Setting switchx27 (internal←location103 switchx27))

8     (Outlet switchx22 (internal←location102 switchx22))

9     (Location boxx17 (location101 boxx17))

10     (Destination boxx12 (bin100 boxx12))


>in
```

```
 un:INVOCATION-42
 Insert a box.

>in

 un:UPDATE-91
 The box's location is updated from input1 to source1.

>show State before or after execution of current statement?after
Final or Incremental knowledge?incremental

1      (OR (= boxx38 boxc108) (Location boxx38 input1))

2      (Location boxc108 Source1)

3      (Location boxx17 (location111 boxx17))

4      (Destination boxx12 (bin100 boxx12))

5      (Outlet switchx22 (internal←location102 switchx22))

6      (Setting switchx27 (internal←location103 switchx27))

7      (= switch1 (switch104 sourcex32))

8      (SourceOutlet sourcex32 switch1)

9      (OR (= bin2 (bin100 boxx12)) (= bin1 (bin100 boxx12)))

10      (Outlet switch1 bin2)

11      (Outlet switch1 bin1)

>justify Fact
Enter fact number ==> 2
```

The user asks for a justification of fact 2, and sees that it resulted directly from the update in the Insert action.

```
Primitive
(Primitive (Update ( un:Location  un:SYMBOLIC-108  un:Source1)
                   un:UPDATE-91 NIL NIL (NOT ( un:Location  un:SYMBOLIC-108
                                                            un:input1))))
```

References

1. Robert Balzer, Lee Erman, Phillip London, Chuck Williams. HEARSAY-III: A Domain-Independent Framework for Expert Systems. Proceedings of the First Annual National Conference on Artificial Intelligence, AAAI, August, 1980, pp. 108-110. A Manual is in preparation by Steve Fickas

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. Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

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

5. Neil M. Goldman. *AP3 Reference Manual.* USC Information Sciences Institute, 1983.

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

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

8. 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.

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

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

11. Warren Teitelman. *Interlisp Reference Manual.* XEROX Palo Alto Research Center, 1978.

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

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

14. David S. Wile. *POPART: Producer of Parsers and Related Tools. System Builders' Manual.* USC Information Sciences Institute, 1981.