The Mechanization and Documentation of Software Specification (A Proposal)[1]

By

Stephen Fickas

Computer Science Department

University of Oregon

May 1984

[1]This proposal was submitted to the National Science Foundation, and has been accepted. The grant supports the author and two graduate students, Jane Laursen and Dana Laursen.

Abstract

The research proposed here addresses the mechanization of software specification; specifically, the construction of formal specifications from a general, domain-independent, implementation-independent language. The current non-mechanized development of such specifications presents several problems:

- *Completeness*
  The large amount of low level detail necessary in a formal specification makes completeness a significant problem.
- *Consistency*
  The development (creation, refinement, modification) of a formal specification must adhere to certain consistency constraints, which are often difficult to maintain manually.
- *Mundane details*
  During the development of a specification, the user must be concerned with both high level strategic decisions and the low level manipulations necessary to implement them.
- *Integration*
  The integration of automated software tools relies on a common data base that contains a rationalized history of the development process. A non-mechanized specification process makes no contribution to this data base. That is, the development of the specification is done outside of the computer (except for the lowest level editing steps) leaving the process informal and undocumented.

The goal of this research is to address each of these specification problems: completeness, consistency, automation of details, and documentation and integration. The research will build on two prior efforts in the area of mechanization of software development: 1) the formal specification language Gist, and 2) Glitter, a system for automating the transformational development of software. The general approach centers on two keys ideas: 1) the effort expended in constructing a complete, consistent formal specification can be at least partially *reused*, 2) development steps can be mechanized, and hence *automated* and *documented*.

1. The Mechanization of the Software Development Process

Recent work in software development has advocated a shift in the way software is specified, implemented, tested, and maintained. This work, as evidenced by [Balzer 81, Darlington 81, Green 79, Rich et al 79] among others, proposes that not only the *products* of the software life-cycle be captured by the machine, but the life-cycle *processes* as well. In particular, machine-based tools should support the user in his construction of specifications, his development of those specifications into a working system, and his maintenance of the system in the face of changing specifications. Such an approach provides at least two major advantages:

- Potentially large portions of a process can be automated. This allows the user to concentrate on decision making and guidance, leaving the machine to handle low level manipulation and analysis.
- The same portions can be formally documented in a machine usable fashion. This allows the construction of a corporate, development data-base, which defines the evolution of the system and can be shared among the various development processes.

In my thesis I demonstrate how automation can be used as a lever for increasing software productivity [Fickas 82]. The research proposed here is a continuation of the automation effort: by using and extending previous results in the area of mechanizing the software *design* and *implementation* processes, it proposes to mechanize the software *specification* process. That is, the proposed research attempts to bring the specification process into the machine, allowing it to be automated, documented, and hence integrated with development and maintenance tools.

The remainder of the proposal is laid out as follows: the remainder of this section discusses the role of formalization in mechanized tools; Section 2 presents the overall research goal, mechanizing the software specification process; Section 3 introduces the Glitter problem solving system; Section 4 discusses the role Glitter might play in mechanizing software specification; Section 5 presents the two key ideas which tie the proposed work together, reuse and refinement; Section 6 shows a small example of the proposed system in action; Section 7 provides a summary of the proposal.

1.1. The need for formalization

Much current software practice relies on both informal products and processes. This informality is a major cause of the software problem:

- Any software specification is likely to have some combination of missing, imprecise or inconsistent requirements. This is exacerbated in an informal specification in which a) the writer expects certain requirements to be of a common-sense nature, and hence not necessary to record, and b) no "specification validation" process is available. For the former in particular, English (or pseudo-English) specifications seem to suffer from many of the problems associated with belief-based Natural Language (mis-)understanding. Specification errors may not manifest themselves until deep into the development process: imprecision and inconsistency during implementation, omission during testing or delivery. Because of this, they are often the most difficult and costly to correct.
- Producing a target program from an informal specification is an error-prone task, and one difficult to control. The resulting program is unlikely to meet specifications fully or be resource-optimal.
- Although standards exist, documentation of large evolving systems is often useless. Frequently this is because of after-the-fact documentation, often by a third party distinct from developers and maintainers. Even when produced during development, documentation is generally incomplete, hard to understand, and difficult to relate to the actual system. As a system is changed, documentation is rarely updated correspondingly.
- A large part of system cost is devoted to modifying software to correct bugs and to meet changing specification requirements. The informality of specification, implementation, and documentation combine to make this both a costly and onerous task.

A key to the machine-based approach is the attempt to bring formalism to the above life-cycle processes. Formalism must start with the definition of a formal software specification language. Such a specification language acts as the key component in an integrated environment:

- An operational specification language (one with an interpreter) can be executed, albeit slowly. Hence, testing can be carried out at the specification level, allowing the user to get immediate feedback.
- A formal specification provides the necessary input to a mechanized software development process.
- A formal specification, in conjunction with a transformation paradigm, makes it feasible to carry out maintenance on the specification as opposed to the target program. The potential savings here are enormous. Instead of a maintainer dealing with highly optimized code which is likely undocumented, he or she deals directly with the problem statement. Both debugging and enhancements occur at the specification; a new program is obtained by re-development.

There exist different formal software specification languages, generally one for each research effort in the area. While each has its virtues and supporters, my research has and will continue to focus on the Gist specification language [Goldman and Wile 79]. The next section provides an overview of Gist.

1.2. The Gist Specification Language

The Gist specification language is a general purpose, domain-independent problem statement language. It provides three major features. First, no valid implementation need be ruled out; the Gist language does not inherently force certain design decisions to be made during specification. However, neither does it enforce any notion of appropriate abstraction level, i.e., it is up to the specifier to choose the correct abstraction level at which to state the problem.

Second, Gist is operational. An interpreter exists for Gist (currently a subset of the language); a Gist specification can be executed to provide specification "validation".

Third, a well defined semantics exist. This supports other useful tools such as property provers and English paraphrasers.

To provide the specifier with an implementation-free language, Gist includes the following components ([Fickas82, Feather&Londond 82] provide a more detailed, example-driven presentation):

- Relational model of information. Information in Gist is modeled by typed objects and relations among them. Certain built-in relations are provided, e.g., sets, sequences.
- Historical reference. Information can be extracted from any past state. A construct such as "asof everbefore" allows a specifier to describe *what* past information is needed without concern of *how* it is to be made available.
- Constrained non-determinism. Non-determinism frees a specifier from making premature selection or control decisions. Constraints rule out invalid selections or control paths. Together, they form a powerful specification technique.
- Data-directed process invocation (Demons). Gist Demons free the specifier from identifying all locations where an event calls for the invocation of a process; the event can be made part of a Demon trigger and the process the Demon body.
- Derived relations. A derived relation allows the specifier to state, in one place, a derivation (i.e., an invariant among several relations), and use the derived information throughout the specification.

In combination, the constructs above make the modification of a Gist specification a difficult problem: a trace of the effects that a newly added (or deleted or modified) specification construct will have on the rest of the specification must be made. This includes identifying places where the specification is no longer consistent, and showing that the change does not incorrectly rule out all possible solutions. This checking is generally complex, and when done manually, likely to lead to errors.

2. Research goal: the mechanization of the specification process

Gist is one part of a larger integrated, machine-based development system called TI (Transformational Implementation) constructed at USC/Information Sciences Institute [Balzer et al 76, Balzer 81]. The development of software using the TI model roughly follows the process below:

Step 1.
The human *specifier* manually constructs a Gist specification. The specifier acts as translator, receiving a problem description from a human *domain expert*, and producing a Gist specification.

Step 2.
The specification is executed by the system and results are compared with the domain expert's intent. Discrepancies lead to changes in the specification.

Step 3.
The specification is mapped, by the human *program developer*, into a target language. The developer uses correctness-preserving transformations, applied by the system, to move from specification to target language.

Step 4.
The target language is "compiled" into a production programming language. This process is generally viewed as outside the purview of the TI system. It is expected that sophisticated, emerging state-of-the-art work in program generation systems will handle the compilation.

Step 5.
The program is run to verify performance constraints. Note that debugging has previously taken place during specification.

Step 6.
Given a requested change by the domain expert, the specification is modified by the specifier and steps 2 through 5 are repeated.

I will refer to the above as the *base-line model* of TI software development. Other papers argue for its power and provide the specifics glossed over above: [Fickas 82, Swartout and Balzer 82, Wile 81]. Its major deficiencies are listed below.

- The transformational development of the Gist specification into the target program is a complex and tedious task. This is not surprising since the transformation process in TI is equivalent to the design and implementation processes in the traditional life-cycle model, both of which are at the heart of what is considered "program development". The majority of the TI developer's time is spent on searching the large transformation catalog, analyzing applicability conditions and applying low level transformations that are of a sub-goaling nature. A relatively small proportion of time is spent with the high level conceptual problems of development.
- The development of the Gist specification is also a complex and tedious task in the TI model. Several factors, each tied to the lack of mechanization, contribute. First, the dark side of formalization becomes apparent. The plethora of mundane requirements that a *human* writer can leave implicit as common-sense in an informal specification -- requirements that he assume a *human* writer can induce -- must be provided in a formal specification. These requirements must be generated anew for each new problem specification. Completeness becomes a problem.

  Second, modification of the specification, while not required to be correctness-preserving, is required to be consistency-preserving. Adding, deleting or changing specification constructs generally has non-trivial ramifications as regards the consistency of the remainder of the specification. While the base-model supplies a rudimentary formalization of the process of mapping a specification to a target program (i.e., program transformations), no such formalization exists for the specification process, i.e., the development of the specification is undocumented and hence not integrated with the base-model. Consistency checks are left to the user.

  Finally, not only must the specifier provide the overall guidance in developing a specification, he or she must also provide the often low level manipulations necessary to bring about a desired state.

- Maintenance in the base-line model is a two step process: 1) modification of the specification, 2) re-development. Both steps are currently manual and non-integrated. That is, there is no formalization of the specification modification process, nor any support in re-developing a target program. In particular, there is no reuse of the original development during re-development.

My thesis [Fickas 82] addresses the first of these problems, the formalization, automation, and documentation of the transformational development process. The second, that of mechanizing the software specification process, is addressed by this research proposal. The third, that of automatic maintenance, remains open (however, see [Wile 81] for some preliminary work in this area). The research proposed here on specification will build on my thesis work in the area of software development. I will first present a synopsis of my thesis work, and then show how it will be used as a foundation for the automation and documentation of the specification process.

3. A Problem Solving View of Software Development

In the previous section, I presented the problems of using the base-line TI model for transforming a Gist specification into a target program. My thesis addresses these problems by automating and documenting a transformational development. In particular, it is based on the following proposition: the development of software, using program transformations, is amenable to an AI Problem Solving approach. That is, a set of development *goals* can be defined, a set of *methods* for achieving those goals can be defined (the program transformations are a subset of such methods and form the leaves of the planning tree), and a set of *selection rules* can be defined for choosing among alternative methods. To support this proposition, I constructed a system, Glitter, which incorporated the goals, methods, and selection rules necessary to handle two moderate transformational developments. Glitter was able to produce a large portion (90%) of these developments automatically. In places where the user was required to provide guidance, it was generally of a high level nature, i.e., Glitter took care of the many mundane steps found in a development.

It may be helpful here to look at a particular example of Glitter in action. Suppose we are given the description of a postal package router. Packages enter the router, slide down chutes, through switches, and into bins. The problem is to implement a controller for the router's switches. This problem was chosen as a study case for specification in Gist because of its interesting features: parallelism, incomplete information (e.g., the location of a specific package within the router is not explicitly known), historical information (the order packages entered the router), errors (misrouting of packages), sensors (event triggered activity). The details of the Gist specification of the router and its mapping can be found in [Fickas 82]; here we will need only a small abstraction to illustrate Glitter's functionality.

Part of the Gist specification for the router contains the following definition for the sequence AllPackages (a simplified version of actual Gist syntax will be used for readability):

```
Gist Sequence: AllPackages
    Definition: {package || located-at[package, entrance]} ordered  by entrance time;
```

That is, the sequence AllPackages is the set of packages that have ever been located at the entrance, ordered by their time of arrival. This definition makes use of several Gist constructs that are not found in the target language. In particular, the sequence is *derived* from an implicit set and a reference to past events (i.e., entrance time). The developer must transform the sequence definition into a form acceptable in the target language. Glitter provides the *Map Away* goal for stating this goal:

```
>User: Map Away derived sequence AllPackages
```

The Map Away goal will be achieved when AllPackages no longer exists as a Gist form, i.e., either it has been implemented in terms of the target language or has been deleted (we will see that a combination of both is possible). Glitter now finds all methods which can achieve the goal. Among these are the following two:

Method Delete-a-construct
  Achieves: Map Away construct C
  Actions: Delete C

Method Maintain-a-sequence
  Achieves: Map Away derived sequence S
  Actions: Incrementally Maintain S

Both of the above methods perform a problem reduction: replace the current goal with some other goal (both Delete and Incrementally Maintain are development goals defined by Glitter). The next step is to choose among the two. The method Delete-a-construct is attractive on the surface: "if you want to get rid of a specification construct, try deleting it; it may not be needed". The question is whether the sequence AllPackages can be effectively deleted. It is at this point that Glitter's selection rules come into play. One such rule says:

IF you are trying to Map Away a sequence S AND
  only the last (first, Nth) element of S is ever referenced
THEN it is likely you can delete S and replace it with a single element

Glitter runs all such rules, and by combining evidence, trims and orders the competing methods. If the method Delete-a-construct was chosen, then the remainder of the problem solving would involve finding all references to AllPackages and replacing each with a single element. If Maintain-a-sequence was chosen, then an explicit sequence would be defined and specific maintenance procedures built to add packages at the appropriate time (i.e., entry time).

The final outcome would be the application of a primitive set of program transformations to achieve the Map Away goal. That is, the developer states his or her high level development goal, and the system figures out the necessary strategies and transformations (does the planning) to achieve that goal. In the actual development from which this is taken, Glitter chose to delete the sequence. To achieve the user's Map Away goal, Glitter produced 45 development steps automatically, 12 of which were primitive program transformations and 33 problem reductions or restatements.

4. A Problem Solving View of Software Specification

The assumption underlying the Glitter system is that Problem Solving techniques can be applied to automate and document the transformation of a Gist specification into a target program. A premise of the research proposed here is that the same techniques can be used to automate and document the specification process. The process of developing a specification has much in common with developing a target program. In particular, once an initial skeleton specification is constructed, much of the remaining task is the refinement (transformation) of this into the final specification. Of course there are differences:

- The TI program transformation process starts with a formal machine-readable object, the Gist specification. The specification process starts with some set of informal requirements in the domain expert's head. The specifier translates these into an initial Gist program. This translation process is not formalized and remains undocumented (as does the refinement of the initial specification into the final version). Previous research in this area has attempted to move directly from the domain expert's requirements description (in restricted English) to final specification, bypassing the specifier (see for instance [Balzer et al 78]). I propose to keep the specifier, but ease his burden, i.e., provide a specification assistant.
- A program-transformation is correctness preserving; a specification-transformation need only be consistency-preserving. Because the developer is groping for the user's intent, the specification-transformations must allow the same flexibility as a structured editor. That is, the specification

developer must be free to change the problem specification. However, just as in mapping methods, specification methods must contain preconditions: in the former they center on equivalency, in the latter on consistency and correctness. McCune looks at some of these issues in regards to the PSI program development system [McCune 79].

The use of the Glitter model for specification development rests on the premise that there are a tractably small set of specification development goals that a specifier may state, and a finite set of methods for mapping those goals onto primitive operations. Part of my research will be the identification of those goals and methods, and the incorporation of them into a system for developing Gist specifications. Section 5.5 shows, through a hypothetical dialog, how such a system might be used in developing a specification.

5. Proposed Research

My research plan is broken into two parts: 1) the construction of the initial specification, 2) the development of this into the final specification. This binary cleavage of the problem is done for pragmatic reasons: it allows two difficult problems to be studied separately. However, schema retrieval and schema refinement cannot be viewed as totally independent. It is possible that the refinement of one schema will call for the retrieval of another. I rule this type of interaction out only in the initial stages of the research; as problems in cataloging and refining schema are worked out, I expect strategic problem solving issues to come to the forefront.

5.1. Two key ideas: reuse and refinement

A number of researchers have noted the problems associated with building complete, unambiguous formal specifications. Approaches have been varied:

- Allow the domain expert to talk directly to the machine in Natural Language. The SAFE [Balzer et al 78], PSI [Green 79] and NLPQ [Heirdorn 76] systems each embraced this approach to some degree. Each had limited success on small problems. It appears that a large amount of research remains to produce such a system for an interesting set of domains.
- Construct domain-specific languages which allow a domain expert to write specifications using objects and operations tailored to his world. The Draco [Neighbors 80] and ECL [Cheatham 81] systems, among others, take this approach. Both have had impressive results in a small number of domains. In particular, the DRACO system allows a domain-specific language to be catalogued, and hence reused by future specifiers.
- The Programmer's Apprentice (PA) project at MIT [Rich et al 79, Waters 82] provides a programmer with general building blocks or plans for constructing programs. Currently, the semantics of these plans have allowed low to medium level programming concepts to be represented. The development paradigm is one of a programmer choosing a general plan and then filling in details until a LISP program is produced. As of now, the PA has no means of formally specifying a problem ([Fickas 82] discusses the differences between the Programmer's Apprentice and TI approaches to software development.).
- Allow the user to quickly see the results of the specification. Rapid Prototyping is generally used as the moniker for this work. Gist, for one, is operational and hence falls in this class [Cohen et al 82].

There are two kernel ideas provided by the above work from which my work will build:

1. *Reuse of analysis*: the DRACO system recognizes that the process of defining the objects, operations and constraints of a domain is time consuming. Redoing domain analysis for each new specification is wasteful, error-prone, and hence costly. DRACO attempts to reuse analysis by producing domain-specific languages as a by-product of analysis. These languages can then be reused in other problems by other analysts/specifiers.
2. *Retrieval and Refinement*: the Programmer's Apprentice allows program plans to be constructed and then reused as building blocks of other programs. The process is one of retrieving an abstract plan

and then refining it to concrete form by filling in slots.

In the proposed system, reuse will be accomplished by abstracting and cataloging Gist specifications; development will consist of retrieving an abstract specification and refining it into the desired form.

5.2. A library of Specification Schemata

The specifier will have a library of Gist specification schemata from which to choose. These shemata will be initially indexed by domain (other types of indexes will be considered at a later date). Each shemata will contain abstract to concrete, partially to fully instantiated Gist constructs useful for stating problems in the indexed domain. An example may be useful here. Suppose that the specifier wished to specify a *routing* problem:

>User: Retrieve routing schemas

The following routing schemas are available:

1) Physical-Router - routes generated physical objects
to m destinations

2) TravSalesman-Router - routes 1 physical object through
each of k locations

3) ...

```
>User: Display Physical-Router

    Name: Physical-Router
    Description: routes generated physical objects
                 to m destinations
    Domain-index: routing
    Parameters: number of destinations m
    Schema:

      type  physobj
      type  location with subtypes (destination, entrance);

      relation  located-at[physobj,location]
      relation  destination-of[physobj, destination]
      relation  connected-to[location, location]

      constraint  ObjectsCantBeInSamePlace
            definition
             ~ exist o1|physobj, o2|physobj, k|location
                          || located-at[o1, k] AND located-at[o2, k];

      constraint  ObjectCantBeInDifferentPlaces
            definition
             ~ exist o|physobj, k1|location, k2|location
                          || located-at[o, k1] AND located-at[o, k2];

      demon  CreateObject(o|object, e|entrance)
          Trigger: RANDOM
          Response: create o|object;
                    assert located-at[o,e];

      demon  SignalArrival(o|object, d|destination)
          Trigger: located-at[o, d] AND destination-of[o, d]
          Response: ?correct-arrival-response?;

      demon  MoveObject(o|object, l1|location, l2|location)
          Trigger: located-at[o, l1] AND
                   connected-to[l1, l2] AND
                   ?condition?
          Response: located-at[o, l2];
      ...
```

The above schema contains objects, operations, and common sense knowledge about the routing world. The portion shown above is just one part of the entire routing schema; the majority of the schema has been elided. The full schema contains all of the constructs useful in specifying routing problems.

Suppose a specifier decided to use the routing schema to specify a particular routing problem. After retrieving the schema, the next step would be to refine and tailor the schema to fit the problem at hand. This might involve deleting skeletal constructs which overspecify the problem (e.g., the constraint that two objects can't be in the same location may be an overconstraint), changing constructs, refining constructs, filling in slots (bracketed by question marks above), and adding constructs to fill out the specification. The key point is that the above schema has captured some part of the analysis needed to build specifications in the routing domain. The specifier can reuse this analysis for his or her particular problem.

Note that while there is support for the basic two step process of constructing (retrieving) a skeleton solution and then refining and debugging it [Sussman 75, Larkin 81], others have argued that the software specification process must follow a more evolutionary model, constantly switching between specification and implementation [Swartout and Balzer 82]. It is this switching that often proves so costly. The need to modify the specification after the commencement of implementation can be caused by unforeseen implementation tradeoffs and limitations, as well as enhancements requested by the user after the system is delivered. These types of problems are difficult to predict during specification, and are beyond the scope of this work. However, another major cause of development backtracking is addressed by this proposal, that of incomplete specifications. Because of the complexity of the specification process, the *manual* production of a complete specification, one containing all of the necessary constraints, objects, and operations, must follow a basic cycle of specification/implementation/debugging. By starting with a more complete (but abstract) specification, iterations through this costly cycle can be reduced.

5.3. Schema Refinement: Glitter Reapplied

The Problem Solving model introduced by Glitter will be used as the basis of specification refinement and modification. This requires focusing on three areas:

(1) *Goals.* A set of specification goals must be defined. Each will represent a type of abstract operation we might want to carry out on a specification. For instance, we may wish to further *constrain, augment, refine, define, consolidate, generalize,* or *delete* various portions of a specification. These types of goals have been found useful in the small set of examples studied so far. It is expected that others will be needed. In particular, the results of other research efforts in documenting the specification process will have a favorable impact here. For instance, Wile's PADDLE system [Wile 81] is capable of documenting the (informal) goals that a developer goes through in creating a program. Glitter benefited from this work by using this documentation as a guide in constructing its formal development goals. Any similar documentation of informal specification goals is expected to have similar payoffs in the proposed system.

(2) *Methods.* A set of methods for achieving specification goals must be defined.The system's methods allow goals to be mapped onto primitive editing operations (the "transformations" or primitive operators of the specification world). The crucial question here is one of generality. The use of Glitter to map specifications into implementations employed only domain-*independent* methods. It is expected that some mixture of domain-independent and domain-dependent methods will be needed in developing specifications. Clearly, the richer this mixture is on the side of domain-independence, the more concise and efficient the system becomes (section 6 gives examples of several domain-independent methods). However, the use of domain-dependent methods is not viewed as inherently evil, and useful ways to and manage them will also be studied.

(3) *Selection rules.* A set of selection rules for choosing among alternative methods must be defined. During program development, Glitter's selection rules were used to automatically find, count, and analyze low level Gist constructs; the user was relied on to supply more insightful analysis. We expect that while new rules will be needed in specification development, many of Glitter's existing analysis routines will be reusable. We plan to use the same interactive partnership approach to reason about problem solving.

Glitter will be used as the foundation of a problem solving system for specification development. By removing the portions particular to software development -- the development goals, methods and rules -- the Glitter machinery can be reused (in fact it appears that some of Glitter's old goals and methods will remain useful in this new problem solving domain). In particular, the following Glitter components currently exist: a goal writing language, a method writing language, a selection rule writing language, and the basic problem solving control processes necessary to turn the system over. Also in place are the mechanisms that allow a problem solver to explore various solution paths, e.g., suspending the current path, spawning a new path, resuming a suspended path.

5.4. Robustness

The Glitter model is based on evolving competency: the goals, methods, and selection rules are expected to be incomplete, but gradually to become richer as experience is gained. This requires two system features. First, the user should not be locked into using only the pre-defined goals and methods of the system. If a goal or method is missing, the user should be able to bypass normal operation, and continue processing. In Glitter, this was accomplished by allowing the user to define ad hoc goals and methods as they were needed.

Second, the system should help identify areas where its knowledge is weak or missing. In Glitter, this was accomplished by monitoring the user's actions. For instance, the use of an ad hoc goal or method as discussed above lead Glitter to document the problem solving context in which it occurred. This is based on the assumption that use of an ad hoc technique signals a missing piece of knowledge. It is likely that the ad hoc technique can be generalized for inclusion in the system's permanent catalogs.

5.5. Documenting the problem solving process

A byproduct of the Glitter problem solving process is a history of the user's goals, the methods competing to achieve those goals, the method chosen, and the reason for choosing it (and rejecting the others). Further, Glitter maintains a dynamic representation of the exploration carried out by the user. This allows the user at any time to:

- Display some or all of the current paths through the planning space.
- Move to any problem solving state.
- Ask to see a) the method set competing in a problem solving state, b) the selection rules that have fired, and c) their effect on the method set.
- Choose any method from the competing set (i.e., spawn a new branch).

This documentation forms the basis of the development data base crucial to an integrated approach to software development. By using Glitter during the development of the target program, the rich planning structure of that process was brought inside the machine; by using the Glitter framework in specification, I expect to reap the same benefits.

6. A hypothetical, specification development

Below is a hypothetical example of the proposed model in use. This example is extracted from an informal specification development transcript kindly provided by Martin Feather. Note that this development started from scratch. Hence, much of the specification development up to the state below has been concerned with goals that add objects and operations, and fill-in details, i.e., (re)doing analysis. Given a good specification schema from which to start, most of the objects, operations, and constraints would be predefined, leaving the specifier to fill-in slots and provide tailoring goals.

The Elevator Problem

The problem is one of specifying an elevator controller. The portion of the specification shown below represents some intermediate development state in which the major objects and operations have been specified. Note that this specification might have started as a schema transportation abstraction, i.e., something that captures information about transportation domains, e.g. buses, elevators, planes, taxis, etc. Such a schema would contain types of conveyors, passengers, properties of conveyors (containership), locations, destinations, etc.

```
type  elevator
type  floor with subtypes (Top, Bottom)
type  direction = {Up, Down}
type  passenger

relation  E-located-at[elevator, floor]        (* an elevator's location *)
relation  P-located-at[passenger, floor]       (* a passenger's location *)
relation  destination[passenger, floor]
relation  inside[passenger, elevator]

constraint  elevator-is-container
  definition forall p|passenger, e|elevator
                  inside[p, e] implies  E-located-at[e, *] = P-located-at[p, *]

demon  enter-elevator(p|passenger, e|elevator, f|floor)
  trigger: E-located-at[e, f] AND
           P-located-at[p, f] AND
           ~ inside[p, e]     AND
           ~ destination[p, f]
  response: assert inside[p, e]

action  move-elevator-up(e|elevator)
  definition update E-located-at[e, f] to E-located-at[e, f+1]

action  move-elevator-down(e|elevator) ...
```

There are several places where the specifier may focus his or her attention:

- The relation "inside" is under-constrained. A passenger should not be allowed inside more than one elevator. Note that the constraint ObjectCantBeInDifferentPlaces from the routing domain (see 5.2) would be useful here.
- The move-elevator-up action is under-constrained. Elevators may currently move through the roof. Since this specification is being constructed from scratch, we can assume that the specifier either forgot this detail or is ignoring such details until a later time. If, on the other hand, the specification had been retrieved from the catalog of skeleton specifications, then we might expect such a constraint, in some abstract form, to be present, i.e., many types of conveyances have route boundaries that they travel between, but not through.
- An elevator, as defined above, is overly abstract. Elevators have doors, switches, phones, etc. At least some of these must be made explicit.
- Some elevators have lock-and-key systems for traveling to certain floors. This would require further constraints on elevator movement.

Here we will look at two of these, constraining an elevator's upward movement, and refining an elevator to have doors.

```
>User: Constrain the action move-elevator-up
```

Among the methods for achieving this goal are the following two:

Method Constrain-action-locally
  Achieves: Constrain action A
  Actions: Add local constraint ?C to A

Method Constrain-action-globally
  Achieves: Constrain action A
  Actions: Add global constraint ?C to the specification

The first method will add a local constraint to the action; the second will add a global constraint to the specification. Since either type of constraint can be equivalently transformed to the other, selection here is a matter of style. If we expect the same constraint to be needed in other locations, then the choice would be a global constraint. Otherwise, the local constraint provides a focus that is easier to understand. One or more selection rules would be the repository for this style information.

A note on domain-specific methods: assume that the system chooses Constrain-action-locally. The constraint ?C must now be defined (the slot filled-in). Initially, the system will rely on the user to supply it. However, since the overall goal is to automate as much of the specification process as possible, we might ask how the system could be of more assistance here. A "smarter" method may take note of the action being taken (effectively, the increment of a variable) to provide all or some portion of the constraint. An even smarter method might take note of the domain we are working in (transportation) and the objects and operations we are working with (the movement of conveyances). Such methods move away from the domain-independent methods found in Glitter. At least one interesting approach would be to attach such methods to specific schemata, loading them when the schema is retrieved (the DRACO system [Neighbors 80] uses a similar technique for loading domain-specific optimizing transformations).

Assume that the system has chosen to add a local constraint. The specifier will be asked to supply it (note the system will take care of the syntactic details of where to insert it):

action move-elevator-up(e|elevator)
  definition
    Precondition: <mouse position>
    update E-located-at[e, f] to E-located-at[e, f+1]

The specifier may now enter the needed condition:

¯ E-located-at[e, top]

Next the specifier may want to move towards a more detailed view of the real world. For instance, other parts of the specification will need to have a more refined view of an elevator.

>User: Refine type elevator

The system finds, among others, the method below.

Method Refine-object-type
  Achieves: Refine type T
  Actions: Define subtypes of T
           Define parts of T

The above method reduces the Refine goal to two sub-goals, both involving the Define goal. Other methods trigger on the Define goals until actual operations are generated. In this case, the final outcome might include the definition of the subtypes Local and Express, and the subpart door. The problem

solving process (along with user interaction) for achieving the refine goal would produce the following (changes are flagged with |):

```
| type  elevator = {local, express}
  type  floor with subtypes (Top, Bottom)
  type  passenger
  type  direction = {Up, Down}
| type  openorclosed = {Open, Closed}

  relation  E-located-at[elevator, floor]        (* an elevator's location *)
  relation  P-located-at[passenger, floor]       (* a passenger's location *)
  relation  destination[passenger, floor]
  relation  inside[passenger, elevator]
| relation  part-of[elevator, door]
| relation  position-of[door, openorclosed]

  constraint  elevator-is-container
    definition forall p|passenger, e|elevator
                    inside[p, e] implies  E-located-at[e, *] = P-located-at[p, *]

  demon  enter-elevator(p|passenger, e|elevator, f|floor)
    trigger: E-located-at[e, f] AND
             P-located-at[p, f] AND
|            position-of[part-of[*, elevator], open] AND
             ~ inside[p, e] AND
             ~ destination[p, f]
    response: assert inside[p, e]

  action  move-elevator-up(e|elevator)
    definition precondition ~ located-at[e, Top] AND
|                             position-of[part-of[*, e], closed]
               update E-located-at[e, f] to E-located-at[e, f+1]

  action  move-elevator-down(e|elevator)  ...
```

In the above step, the system plays a more powerful role. The refinement of an object will likely have ramifications throughout the specification. We are taking what the rest of the specification regards as a blackbox (an elevator), and splitting it open (adding doors, etc.). Hence, the methods chosen to refine elevator must address more than the problem of adding subtypes and part-of relations; they must also focus on how the newly refined object now interacts with existing constraint, demon, and action definitions. The changes to enter-elevator and move-elevator-up point this out. The capability to both find all locations effected by a specification modification (a syntactic aid), and assist in changing each to reflect the new viewpoint (a semantic aid) are tow of our major research goals.

7. Summary of Proposed Research

Problems Addressed

(1) A large amount of low level detail is necessary in a formal specification. Much of this detail is of the common-sense variety left out of informal (e.g., English) specifications. The handling of this detail manually by the user is both tedious and likely to lead to incomplete specifications. The cost of an incomplete specification can be high if not caught until the design or implementation stage.

(2) A change to one part of a specification can have complex effects on the remainder. As in completeness checking, consistency checking, if done manually, is complex and error-prone. Consistency errors can have the same high cost as completeness errors.

(3) In developing a specification manually, the user is responsible for providing both high level strategies and the low level steps necessary to carry them out. As with domain analysis details, manipulation details can be tedious and error-prone if carried out manually.

(4) Other development tools rely on information (e.g., rationale, dependencies) about the specification process. Such information currently is not documented, and hence is unavailable.

Proposed Solutions

A system is proposed that has two major components: a) a catalog of specification schemata, and b) a problem solving system for refining a schema into a complete, consistent problem specification. The system addresses each of the problems above:

(1) The library of specification schemata will allow an analyst/specifier to *reuse* past analysis. By defining the fundamental objects, operations, relations, and constraints of a domain, a schema allows the specification process to move away from the current blank-sheet paradigm to one of refine-and-tailor. The completeness problem should be greatly reduced as more complete specifications are passed to the transformation phase, i.e., less backtracking to fix completeness bugs will be required.

(2) The set of development methods will help automate and maintain specification consistency. Subgoalable applicability conditions can be established for each method in the system. In places where automation is not possible, the system can at least ask the right questions of the user. This approach of automating where possible (generally low level details) and querying the user when automation fails has been used successfully in Glitter.

(3) The problem of dealing with the mundane details of specification modification can be handled by relying on the machine to supply the necessary steps. The Glitter system filled this role in the software transformation process by supplying the user with a set of program development goals; the system mapped these goals onto a set of primitive operators (correctness-preserving transformations). The same approach will be used for specification development. Here the user will supply specification goals, and the system will map these goals onto consistency-preserving operations.

(4) A by-product of the Glitter system is a machine-usable history of the planning and exploration space generated during problem solving. This history allows the specification process to be integrated into the overall machine-based development model.

The proposed system will center on two key ideas: reuse and refinement. The library of specification schemata will provide the reuse through storage and retrieval of past analysis efforts. The Problem Solving system will provide the means of refining the skeleton specification into the desired final form.

Potential Difficulties

While the Glitter model provides a sound foundation, the proposed work explores new ground with the attendant risks.

(1) *Schema library.* Other research has shown that is is possible to construct domain-*dependent* specification languages and then reuse them (see for instance [Neighbors 80]). The work here explores the reuse of specifications written in a domain-*independent* specification language. A potential problem is that useful schemata may have to be tied to overly-specific domains, e.g., a schema that only handles elevator problems. My preliminary work in cataloging schema from the transportation domain *does not* support this; a single transportation domain can be used in a wide range of sub-domains. However, there is another mitigating item: part of my current work is exploring the use of composable schema components from which to construct a specification. In this way, a specification for a domain new to the system can be built by taking parts from existing schemas in other domains. This work looks quite promising.

(2) *Problem solving.* The difficulty here is the use of general specification development goals and methods on domain specific problems. Do human specifiers use consistent goals across domains? Can a domain-independent set of methods be found for achieving them? If not, the problem becomes the management of the potentially large amount of domain-specific knowledge the system must possess.

In summary, a reuse and refinement approach is planned, which will produce large payoffs in specification mechanization if successful. The potential difficulties with this approach are recognized, and means of modifying or extending the approach to meet them have been included in the proposal.

8. Research Plan

The first priority of this research will be the construction of a prototype system. This work has already begun. Current research focuses on the following components:

(1) A language for representing domain abstractions. Initial use of Gist as the domain representation language proved awkward. Our current work centers on a domain language that mixes object-oriented constructs with Gist constructs. We expect that the new language will compile to Gist.

(2) A small library of schemata. Because I have analysis/specification experience in the area of scheduling and transportation, I have chosen to look at problems from these domains first.

(3) A problem solving system. Included will be a set of goals for refining a specification schema. The method catalog will contain methods for mapping goals to operators. A set of rules will be defined for selecting among alternative methods.

9. References

[Balzer 81] Balzer, R. (1982)
Transformational Implementation: An example.
IEEE Transactions on Software Engineering, 7(1):3-14, 1981

[Balzer and Goldman 79]
Balzer, R., Goldman, N.
Principles of good software specification and their implications for specification languages.
In Specification of Reliable Software, IEEE Computer Society, 1979

[Balzer et al 76] Balzer, R., Goldman, N., Wile, D.
On the transformational implementation to programming.
In Second International Conference on Software Engineering, 1976

[Balzer et al 78] Balzer, R., Goldman, N., Wile, D.
Informality in program specifications.
IEEE Transactions on Software Engineering, 4(2):94-103, 1978

[Balzer et al 82] Balzer, R., Goldman, N., Wile, D.
Operational specification as the basis for rapid prototyping.
In Proceedings ACM SIGSOFT Software Engineering Symposium
on Rapid Prototyping, 1982

[Cheatham 81] Cheatham, T.
Program refinement by transformation.
In Fifth International Conference on Software Engineering, 1981

[Cohen et al 82] Cohen, D., Swartout, W., Balzer, R.
Using symbolic execution to characterize behavior.
In Proceedings ACM SIGSOFT Software Engineering Symposium
on Rapid Prototyping, 1982

[Darlington 81] Darlington, J.
An experimental program transformation and synthesis system.
J. of Artificial Intelligence, 16(1):1-46, 1981

[Davis 80] Davis, R.
Meta-Rules: Reasoning about control.
J. of Artificial Intelligence (15):179-222, 1980

[Erman et al 81] Erman, L., London, P., Fickas, S.
Hearsay III: Design and an example use.
In Seventh International Conference on AI, Vancouver, BC, 1981

[Fickas 80] Fickas, S.
Automatic goal-directed program transformation.
In First National Conference on AI, American Association
for Artificial Intelligence, Stanford, 1980

[Fickas 82] Fickas, S.
Automating the transformational development of software.
Computer Science dept, University of California Irvine, 1982
Available as ISI/RR-83-108,109 from USC/Information Sciences Institute,
Marina del Rey, Ca. 90291

[Goldman and Wile 79]
Goldman, N., Wile, D.
A data base specification.
In International Conference on the Entity-Relational
Approach to Systems Analysis and Design, UCLA, 1979

[Green 79] Green, C.
Results in knowledge based program synthesis.
In Sixth International Joint Conference on AI, 1979

[Green et al 81] Green, C., et al
Research on Knowledge-Based Programming and Algorithm Design.
Technical Report KES.U.81.2, Kestrel Institute,
1801 Page Mill Road, Palo Alto, Ca., 94304

[Heirdorn 76] Heirdorn, G.
Automatic programming through natural language dialogue: A survey.
IBM J. Research and Development 4:302-313, 1976

[Larkin 81] Larkin, J.
Enriching formal knowledge: A model for learning to solve problems in physics.
In J.R. Anderson (Ed.), Cognitive skills and their
acquisition, Hillsdale, N.J.: Lawrence Erlbaum, 1981

[London and Feather 82]
London, P., Feather, M.
The Implementation of Specification Freedoms.
Technical Report RR-81-100, USC/ISI, 4676 Admiralty Way,
Marina del Rey, Ca., 90291

[Neighbors 80] Neighbors, J.
Software construction using components.
PhD Thesis, Computer Science Dept, University of California Irvine, 1980

[Rich et al 79] Rich, C., Shrobe, H., Waters, R.
An overview of the Programmer's Apprentice.
In Sixth International Conference on AI, Tokyo, 1979

[Sussman 75] Sussman, G.
A computer model of skill acquisition.
New York: American Elsevier

[Swartout and Balzer 82]
Swartout, W., Balzer, R.
On the inevitable intertwining of specification and implementation.
CACM, 1982

[Waters 82] Waters, R.
The programmer's apprentice: Knowledge based program editing.
IEEE Transactions on Software Engineering 8(1):1-12, Jan, 1982

[Wile 81] Wile, D.
Program Developments as Formal Objects.
Technical Report RR-81-99, USC/ISI 4676 Admiralty Way,
Marina del Rey, Ca., 90291