Deriving Specifications from Requirements

W. Lewis Johnson

USC/Information Sciences Institute
4678 Admiralty Way, Marina del Rey, CA 90292

Abstract
Specification-based software development makes software easier to validate and maintain. Yet specifications of large systems are themselves large, making understanding and validation difficult. One cause for this problem is that specifications and requirements are kept distinct. This paper describes an approach to specification development in which the specification arises naturally through the requirements analysis process. The emerging specification is developed into a complete system description using formal transformations called high-level editing commands. Automated support for this development process within the Knowledge-Based Specification Assistant will be described. This support involves applying high-level editing commands, assisting in the choice of editing commands, and tracking the effects of these commands.

1. Introduction
The goal of automatic programming is to allow people to describe what they want a computer program to do, and have the program be automatically generated on the basis of this description. It has been generally assumed that the program description would take the form of a formal specification [3, 20]. But this raises a fundamental question: how do you know that the formal specification really describes the computer system that you have in mind? Formal specifications are hard to read, and their semantics do not always agree with one’s intuitions. It is necessary to ensure that the specification is validated before it is used as the basis for implementation.

One way of making specification validation easier is to make the specification language fit the way people think about systems as closely as possible [2]. Another answer is to provide tools which assist people in interpreting a specification. For example, symbolic evaluators can be used to execute the specification and look for unintended behavior [4, 5]. Other analysis tools can examine the output traces of the symbolic evaluator, and point out potential problems to the specifier [24]. If the specification is paraphrased into some other medium, such as English, then the new perspective on the specification can lead to the discovery of specification bugs [23].

Such techniques are all useful, and we continue to make use of them in our work with specifications at ISI. However, they are not sufficient when one is developing a specification for a large, complex system. A natural specification language does not solve the problem, because a large amount of detail is required to specify a system, no matter what the language is. Contradictions and inconsistencies are therefore difficult to spot. The output of evaluation and analysis tools becomes hard to understand, because there is so much to evaluate and analyze. Even worse, as specifications get larger, they get further and further removed from the original system.
requirements. System developers first write a large requirements document, and then use it as a basis for writing a large specification. The mapping from requirements to specification is not made explicit.

The Knowledge-Based Specification Assistant project at ISI [16] is developing a knowledge-based system to help specifiers develop specifications, and ensure that they are valid. In our view, the basic problem with most specification validation tools is that they are applied only after the specification is written, and long after the requirements are written. Our approach instead is to get the machine "in the loop" from the early stages of the software development process. The description of the system has a rich formal semantics even at the requirements analysis stage. The Specification Assistant then works with the user to construct the completed specification. It assists in making decisions about how to develop the specification, and then carries out those decisions. As a result, it helps to ensure that the resulting specification is valid.

This paper will discuss how system goals and requirements can be formalized during the early stages software development, and how these descriptions can then be transformed into completed specifications. It will describe how the Knowledge-Based Specification Assistant helps the user carry out this transformation process.

2. Integrating Requirements and Specifications

Our method of specification development employs the specification language Gist [13, 17, 18]. Gist is a wide-spectrum specification language. Using Gist, one describes both the system which is to be implemented and the environment in which the system is to operate. These descriptions consist of definitions of types, relations, procedures, and constraints. Gist specifications describe the behavior of the system and the environment over time. Gist is operational in that sense that one can execute a Gist specification by simulating the processes and state changes for a given behavior.

As is the case for any wide-spectrum language, Gist is intended to permit implementations to be formally derived from specifications by applying transformations [9]. The transformations remove high-level programming constructs such as non-determinism and constraints from the specification. In the process, the specification is gradually transformed into a conventional program.

Goldman first recognized that it also might be possible to formally model the development of Gist specifications [14]. It is possible to describe the development of a Gist specification as a sequence of changes to an initial specification, each of which elaborates the specification in some way. These changes are similar to the transformations that are used for turning specifications into implementations. There remained some open questions as to how such a process might fit into an overall software development paradigm, however.

- How does specification development relate to requirements analysis?
- What information goes into a specification at each stage in the development?
- Is Gist adequate as is for supporting specification development, or should it be extended or revised?

Our work on the Knowledge-Based Specification Assistant has focused on providing such automated support for specification development. The experience gained in this work puts us in a better position to tackle these questions.

2.1. Specifications within requirements

We believe that specification development and requirements analysis should not be separated. As soon as the software developer starts describing a software
system, he or she needs to be able to build formal, operational representations of the system. In fact, specifications and requirements should not even be thought of as distinct; specifications are simply those parts of the requirements that have been formalized well enough to have an operational semantics. Specification and requirements deviate only when the specification is transformed into an implementation.

The requirements analysis process is particularly facilitated if the following is described using an operational language:

- the behavior of entities in the environment,
- and
- the overall requirements of the system, including the requirements that the implemented system and environmental agents together must meet.

These two go hand in hand. By modeling the environment in an operational fashion, the analyst can run simulations of the environment, to determine more precisely what external conditions the implemented system must react to. Such additional precision then allows requirements to stated formally, so that they can be tested against the resulting specification and implementation.

There is strong psychological evidence for the need for operationality throughout software development. Software designers have been shown to perform mental simulations of systems, both during requirements analysis and design [1, 19]. Formal operational descriptions of systems allow the simulation to be performed by machine, resulting in more accurate requirements analysis.

To illustrate the role that operational descriptions at the requirements level can play, we will focus throughout this paper on a particular application domain: air traffic control. An air traffic control system is supposed to assist controllers in tracking and controlling aircraft throughout an air space. Each aircraft is presumed to have filed a flight plan. The job of the controllers is to ensure that the aircraft adhere to the flight plans. During an aircraft's flight it may travel through multiple air spaces, each controlled by a different air traffic control facility. As the aircraft move from air space to air space, control of the aircraft must be handed off from one facility to the next. Within a given facility individual controllers may also hand control off to each other.

In order to understand what requirements an air traffic control system must meet, it is necessary to model in detail the environmental agents that the system will interact with. One must model aircraft filing flight plans and following them, how controllers and control facilities hand off aircraft, and how radars behave. Given such a model, one can then formalize what behavior the air traffic control system should help cause to come about. For example, aircraft should always be handed off to the appropriate controller; exactly one controller should control an aircraft at all times; aircraft should be made to adhere to their flight plans. These conditions are not specifications of the behavior of any particular specification component; in fact, it is not yet clear what role the air traffic control system plays in maintaining these constraints. Nevertheless, they can and must be formally and operationally described, so that the analyst can simulate them in conjunction with the environment, and make sure they are valid.

Not only should requirements be formalized: the process of developing specifications can also be formalized. We have identified a number of high-level editing commands to assist in this process. These editing commands are transformations, similar to implementation transformations, but they are not correctness preserving.

3This problem is one of several that we have studied in the context of the Knowledge-Based Specification Assistant project. Our principal focus has been on two problems, however: the air traffic control problem and a hospital information systems problem.
Instead, they allow the analyst to change the requirements in meaningful ways. Some commands add detail to the requirements, or to the model of the domain; others flesh out the specification of the implemented system. Such commands provide an opportunity for computer-based support of the specification development process. The analyst can develop the specification without introducing clerical errors. The relationship between the meaning of the specification at each point in the development is understandable in terms of how it changed.

This view of specifications and requirements differs from most accounts of specifications and requirements. Requirements languages are usually a mixture of informal and formal descriptions (e.g., PSL [25] and KBRAs representation [22]). The formal component of such descriptions are not operational in any meaningful way; at best, they identify agents and relationships, and trace the data flow between agents, without modeling this data flow over time relationships (e.g., [6, 21]). These representations may characterize the behavior of the system, but they do not support detailed analysis or simulation.4

Our approach also contrasts with most operational specification methodologies, such as those of Zave [27], Terwilliger et al. [28], and Finkelstein [12]. Zave and Terwilliger presume that specifications are developed only through correctness-preserving transformations; the specification is presumed to be correct at the start. Finkelstein has developed a methodology for specification development; however, this methodology does not support operationality until the very late stages.

2.2. High-level and low-level specifications

We call early versions of a specification developed in the manner described here high-level specifications; we call later versions low-level specifications. The following compares the contents of high-level and low-level specifications, to make clearer what process is involved in developing specifications.

The initial steps in the specification development process are primarily descriptive. The analyst must describe what the givens are for the software development problem: how entities in the domain behave, and how they should be made to behave once the software is installed. The purpose of the subsequent specification development process is to validate these descriptions, to make sure that they agree with fact, and to derive a specification for the software system. This derivation process should be driven by analysis of the givens in the specification.

Specifications, both high and low, contain descriptions of the following components:

- entities, relations, and events, i.e., the terminology for describing the domain and the software,
- capabilities of the agents, i.e., what actions they are capable of performing, and
- behavior employing those capabilities.

High-level and low-level specifications differ, however, in the roles of each of these components.

- High-level specifications primarily describe the entities, relations, and events on the environment; they have relatively few terms describing the system to be implemented. A low-level specification fully defines terms to describe both system and environment. System data is described using entities and relations; functional components of systems are described as entities.
- The extent to which agent capabilities are defined in a high-level specification depends upon whether or not the agent is environmental, and how it interacts with other agents. If an agent does not interact

4There are some partial exceptions to this characterization. Sanders's KBRa system allows the analyst to define state-transition diagrams; however, these descriptions cannot describe behavior in much detail, particularly as multiple agents and objects are simultaneously changing state. Fickas' proposed KATE system will allow analysts to define scenarios of system usage; these scenario descriptions are supposed to be operational.
directly with the implemented software, its capabilities can be described in detail. Agents which interact with the implemented system cannot have their capabilities fully specified until the functions of implemented system are specified. Capabilities of the implemented system may not be described at all at first. For example, the capabilities of aircraft can be described in detail in a high-level specification: aircraft take off, land, change course, etc. The capabilities of controllers include issuing commands to the air traffic control system; until the functions of the air traffic control system are specified, the controllers' capabilities cannot be fully described.

In low-level specifications, behavior descriptions are used solely to describe possible behavior of the environment, and allowable behavior of the implemented system. In high-level specifications, different kinds of behavior descriptions are useful. The high-level specification may describe expected environmental behavior rather than possible behavior. If the behavior involves the implemented system, the description might be allowable behavior, or it might be desired behavior. We call restrictions on allowable behavior application requirements, and descriptions of desired behavior application goals. An example of an application requirement is the constraint that no two controllers control the same aircraft at once; this requirement is used to derive the handoff function of the air traffic control system. An example of an application goal is the constraint that every aircraft in the terminal control area be controlled: this constraint may be violated if aircraft go off course.

Figure 2-1 shows some excerpts from a high-level specification for the air traffic control problem, illustrating what does and does not go into a high-level specification. It defines the behavior of aircraft in some detail, in fact much more detail than is included here. The behavior of controllers is only sketched out, since the actual behavior will depend in part on what functionality the air traffic control system provides to the controllers. The air traffic control system itself is not mentioned at all; its functionality will be derived from the stated constraints on aircraft and controller behavior.

High-level specifications are thus lop-sided, incomplete descriptions of systems. They are lop-sided in that they say much about the environment of a system, and little about the system itself. They are incomplete in that they leave out detail, some of which may be necessary in order to produce a full specification. If the specification describes desired system behavior, without describing the capabilities, then it is not satisfiable. The task of the specifier is to develop the high-level specification into a consistent, complete, implementable specification.

Note that the specification development process develops a specification both of the implemented system and its interface. The analyst identifies system capabilities that could achieve this desired behavior, and specifies how those capabilities are employed both by the system itself and by external agents to achieve the desired behavior. Conventional specification and

8In Gist syntax, the symbol | means "is of type". The symbol ? always appears inside some relational expression; it refers to some object for which the relation holds. || means "such that". Thus expected-position(ac,?) is equivalent to the position || expected-position(ac, position). See [17] for a description of Gist syntax.
requirements analysis techniques take the interface as given, and build up from there. Feather has argued for simultaneous development of system and environment specifications elsewhere [10]. The approach described here deviates slightly from Feather's view; here we claim that the system and its interface must be specified simultaneously, but the rest of the environment description can be taken as given. The given part can be used to provide greater guidance for the specification development, and is likely to be reusable from one requirements analysis task to the next.

2.3. Revisions to Gist
As the above breakdown of specification components indicates, we find that the same basic specification constructs are employed both in high-level specifications and low-level specifications. We found, however, that extensions were required in two areas.

- Mechanisms for supporting incomplete specifications were required. The semantics of Gist was originally defined under the assumption that the specification was complete. High-level specifications are incomplete; they describe desired behavior without describing mechanisms for generating that behavior. Even worse, intermediate versions of specifications may contain references to undefined terms. We needed to build computer-based tools which support incomplete specifications. We also introduced operators which can be applied to specification components, to indicate that they are incomplete.

- We needed ways of describing the intended roles of components of high-level specifications, and better characterizing the properties that the resulting low-level specification should have. For example, we needed a way of marking invariant that all aircraft stay on their flight plans, shown above, to indicate that it is an application goal, not a requirement. We needed ways of characterizing desired properties of the low-level specification, such as what data flow is permissible. This characterizations are added as annotations to the Gist specification.

Examples of each of these extensions will arise in the discussion that follows.

3. Going from High-Level to Low-Level Specifications
In what follows, we will look in detail at some of the properties of high-level specifications, and discuss how they are transformed into low-level specifications. The discussion will also serve to illustrate some of the Knowledge-Based Specification Assistant's capabilities for supporting specification development.

3.1. Removing the perfect knowledge assumption
In high-level specifications, we can specify system behavior without regard for what data is accessible to the system, or to environmental agents. In such specifications, agents are described as if they have perfect knowledge of what facts are true of the environment, and what other agents are doing. In deriving low-level specifications, the specifier must make clear what data accesses are permissible capabilities of the system, and design the specification accordingly.

For example, Figure 2-1 has an invariant relating aircraft-position to expected-position. Suppose that we assign this constraint to be maintained by the air traffic control system. We then have a problem: the ATC system cannot observe aircraft positions directly. It must observe radar track positions instead. The specification must be transformed accordingly.

An example of a high-level editing command which removes perfect knowledge assumptions is the command Splice, which is described in more detail elsewhere [16, 11]. This command replaces references to a relation that should not be observable, e.g., aircraft-position, with a new relation track-position, which is computed by the radar.

invariant for all aircraft \ | \
aircraft-position(ac, expected-position(ac, ?)); 
invariant for all aircraft \ | 
 in-flight(ac) => 
 exists c|controller || control(c, ac)
Splice also adds an invariant constraining the track's position and the aircraft's position to be the same. As this invariant is weakened, the specification becomes more realistic (and more complex).

In order to motivate the use of Splice, one must make clear what data is accessible to which agents. We provide the specifier with a means of annotating the Gist specification in indicate data access boundaries. These annotations are represented internally in a Lisp-like notation, e.g.,

\[
\text{(deny-access :resource aircraft-position :accessor (type atc-system) \text{:test-predicate true})}
\]

The specifier can first write specifications assuming perfect knowledge, and then add annotations describing data accessibility. Analysis of how data is used in a specification can then identify agents which access data which should be inaccessible to them; the specifier may then employ commands such as Splice to correct the information flow.

### 3.2. Defining capabilities

Just as it is convenient to describe agents as if they had perfect knowledge, it is convenient to describe them as if they had unlimited capabilities. That is, agents can affect the environment directly, without going through some intermediate agent. A good example here is keeping the aircraft on course. The following Gist demon can be used to describe at a high level what controllers are to do when aircraft veer off course. It states that whenever an aircraft's position is not the expected aircraft position, the aircraft's position should be changed to agree with the expected position. The demon performs the change simply by updating the aircraft-position relation. In the low-level specification, some procedure will have to be defined which will have the effect of causing the aircraft's position to change (e.g., send a course correction request to the pilot of the errant aircraft).

```plaintext
demon correct-position[ac | aircraft] when ~aircraft-position(ac, expected-position(ac, ?)) do update? aircraft-position(ac, ?) to expected-position(ac, ?)
```

The notion of the air traffic control system "updating" the aircraft's position may seem strange. In order for it to make sense, think of updating not as modifying some fact in a database, but as causing some fact to be true in the world. Relations in Gist can be used either to represent data internal to a software system or to represent facts about the world. By ignoring at the high level what capabilities agents have, one can describe what events agents cause to happen, without describing how they cause them to happen. This ability to describe "what" instead of "how" is also what distinguishes specifications from implementations. In this way low-level specifications can be regarded as implementations of high-level specifications [8].

In order to distinguish "updating" from "causing to happen", and in order to guide the refinement of the high-level specification, the specifier must indicate which data is modifiable by which agents. This is done using annotations, as was the case in describing accessibility of information.

The notion of describing effects regardless of whether or not an agent is capable of causing the effects extends beyond the issue of modifiability of data. It is sometimes convenient to describe an effect even if no agent in the system has a method for achieving the effect. Application goals in particular are frequently stated without regard for whether or not they are achievable using stated capabilities. In such cases the specifier may use the achieve operator, a high-level operator that can be applied to Gist definitions. This achieve operator is similar to the achieve operator of Dershowitz [7]. For example, we could use the achieve operator in the correct-position demon as follows:

```plaintext
demon correct-position[ac | aircraft] when ~aircraft-position(ac, expected-position(ac, ?)) do achieve [] postcondition aircraft-position(ac, expected-position(ac, ?))
```

434
The presence of an `achieve` operator indicates that the specification is operationally incomplete; some action must be added which causes the stated postcondition to be true. Pure Gist has fully operational semantics; if a predicate is to be true in a behavior, then the specifier must define some procedure or demon which is capable of making the predicate true. The `achieve` operator violates this condition. Thus each `achieve` operator ultimately leads to a development goal being posted to supply an implementation for the `achieve` operator.

`achieve` operators are sometimes introduced by high-level editing commands to indicate where details must be filled in. The command `Maintain Invariant Reactively` sometimes does this. `Maintain Invariant Reactively` replaces a Gist invariant stating that some predicate is always true with a predicate stating that it should be made true whenever it becomes false. When possible, it uses an `insert` statement or `update` statement to make the predicate true. Suppose, however, that the predicate is not directly assertable. For example, it might be something like

```plaintext
exists c | controller(c) \& control(c, ac)
```

In such a case the command will add an `achieve` operator, and leave it to the specifier to decide which method should be chosen to achieve the predicate.

### 3.3. Application requirements vs. application goals

Some high-level statements should be satisfied to the letter in the implemented system. For example, no aircraft should be controlled by more than one controller at once. We use the term `application requirement` to refer specifically to these inviolable constraints on the system. Other constraints may not be satisfied exactly, or exceptional cases may arise where they are not satisfied at all; we call these `application goals`. We saw an example of an application goal above: the goal that all aircraft adhere to their flight plans.

We will make a distinction here between `application goals`, which are goals which the application must meet, and `development goals`, which describe actions that the specifier wishes to perform in refining the specification. Application goals describe desired behavior of the application; development goals describe individual tasks that the specifier must perform on the road to producing a low-level specification of the desired behavior. For example, the specifier may identify an application goal that the aircraft must never collide. Then during the specification development the specifier may have a development goal of refining the definition of the air traffic control system so as to minimize the number of collisions. Development goals will be discussed later in the paper.

KBSA allows the specifier to annotate each specification component to indicate whether it is a goal or a requirement. These annotations help to guide the process of transforming each goal and requirement into implementable specification. For example, when incorporating the goal that every aircraft follow its flight plan into the specification, a number of editing commands must be applied, including `Maintain Invariant Reactively`. It yields the following: It generates the following:

```plaintext
demon correct-position(ac | aircraft)
  when
  ~aircraft-position(ac, expected-position(ac, ?))
  do update? aircraft-position(ac, ?)
  to expected-position(ac, ?)
```

This demon states that whenever the aircraft's position is not the same as the expected position, update it to be the same as the expected position. Performing such a transformation on an application goal is called `compromising` the goal, because it moves away from complete satisfaction of the goal. Substituting this demon for the invariant is a goal compromise because now the aircraft can go off course; there may be a time delay between when the aircraft goes off course and when the demon corrects it.

Deciding what compromise to make to a goal involves a substantial amount of design effort in general. Certain kinds of goal compromises occur frequently, however,
among them:

- introducing procedures to maintain invariants,
- introducing exceptional cases when goals are not achievable, and
- restricting the freedom of action to agents so that their goals will not conflict.

We are building high-level editing commands into our library to facilitate these sorts of compromises.

We are building monitoring mechanisms into IU3SA to track the fate of application goals and requirements in the specification. By default, any invariant is considered a goal, and can be compromised. However, if the user annotates an expression indicating that it is a requirement, meaning-changing high-level editing commands cannot be applied to the expression.

4. Guiding the Refinement Steps

Most of our development effort so far has been directed toward construction of high-level editing commands to support our specification refinement methodology. However, high-level editing commands are only part of what one would need for highly automated development assistance. We would ultimately like IU3SA to take a more active role in the development process. We can see what role high-level editing commands play by regarding specification development as traversing a problem space. Each state in this space is a partially completed specification. The specifier is attempting to reach some goal state, in which a specification which adequately meets the specifier's requirements has been constructed. High-level editing commands are the operators that move from one state to the next. KBSA's responsibility currently is to execute the operators that move the specification from problem state to problem state. The user must decide whether or not a goal state has been reached, and if not what operator to apply.

We see the process of deciding what high-level editing command to apply as follows.

- The user starts with some expectations of what the goal states should look like. That is, he is able to analyze or validate a specification, and determine whether or not it is complete. These expectations may be general well-formedness conditions that any completed specification of any problem should meet. For example, in a completed specification all terms should be declared. Alternatively, there may be problem-specific expectations: the data of one agent is inaccessible to another agent, or an invariant is an application requirement.

- The expectations are used to identify issues in the specification that need to be resolved. That is, the specifier decides that some particular aspect of the specification needs refinement. For example, the air traffic control system was accessing aircraft positions even though those positions were inaccessible to the system; this erroneous data access was an issue to be resolved.

- Selection of an issue results in a development goal being posted; in other words, the specifier has the goal of resolving the issue. The above issue of the ATC system referring to aircraft-position resulted in a development goal of changing the data path.

- Finally, an editing command is selected to achieve the development goal, based upon knowledge of what kinds of goals each editing command is capable of achieving.

We see in this context that the annotations and operators what we have added to Gist serve to provide information at higher levels in the decision process. Annotations of data boundaries, compromisibility, etc., allow the specifier to state his expectations. Operators such as achieve make issues explicit in the specification. We are now working to provide facilities in KBSA to use this information to help decide what development goals need to be posted.

Our approach to development goals is to classify editing commands according to the type of development goals that high-level editing commands achieve. Here are some of those classes:

- add detail to domain model
- generalize existing domain model
• revise domain model
• extend behavior, leaving domain model fixed
• restrict behavior, leaving model fixed
• revise behavior, leaving model fixed
• rephrase specification, leaving behavior and model fixed
• move toward implementation.

Using this classification, the specifier can decide what kind of change needs to be performed; the system provides a list of possible commands that fit the classification.

The degree of assistance that the Knowledge-Based Specification Assistant can provide is limited by the amount of information that it has about expectations, issues, and goals. Only so much in the way of general expectations can be built into the system; problem-specification expectations must be provided by the user. For example, KBSA cannot tell by itself whether a Gist invariant is a goal or a requirement; the user must annotate it. We are then faced with a tradeoff situation: when will a user find it advantageous to describe his expectations to the system, and when will he prefer to do the analysis himself? As we gain experience in using KBSA to build specifications, we will be in a better position to address this question.

5. Conclusions and Future Research

KBSA's approach to specification development shows promise as a way of allowing specifiers to state their goals for a system as concisely as possible, and build them into a coherent specification. By introducing specification-like representations at the early stages of software development, the common distinctions between requirements and specifications disappear. Instead, the system analyst pursues a methodology whereby desired behavior is described, and a specification is then derived to perform the desired behavior.

By formalizing the steps that analysts go through in turning requirements into specifications, it is possible to provide automatic assistance for specification development. This assistance can take the form of automating the individual editing changes, as well as helping the analyst make decisions as to which editing changes are appropriate to make. Such assistance is provided by the Knowledge-Based Specification Assistant, with encouraging results.
References


10. Feather, M.S. "Language support for the specification and development of composite systems". ACM Transactions on Programming Languages and Systems 9, 2 (April 1987), 198-234.

11. Feather, M.S. Constructing Specifications by Combining Parallel Elaborations. To appear in IEEE TSE.


18. Johnson, W.L. A brief overview of the Gist language. Draft. Also appears as an appendix in [17].


