A Formalization of a Design Process

James Kirby, Jr., Robert Chi Tau Lai, David M. Weiss
Software Productivity Consortium
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

Software design is frequently viewed as an irrational process, wherein a design can be rationalized after it has been completed [8]. Nonetheless, it is useful to try to formalize the description of the process so that one may understand it better, so that one may provide automated support for it, and so that one may provide guidance to software designers. This paper describes a formal model of a variation of the design process described in [8].

Our model of the design process is a state model; it permits progress in design to be characterized as transitions among states. The model has two levels. The lower level is based on the states of the artifacts produced during the design process; we call such states artifact states *(A-states). In the upper level, states are defined in terms of artifact states, and are augmented with descriptions of activities and the roles of people who may perform activities. Activities include sequences of operations and analyses that can be performed on artifacts within the state. We call the augmented states in the upper level* process states *(P-states). This paper describes the state model, the questions we expect to be able to answer using it, and how we are using it as a guide for the automated support we are developing.*

Keywords: software design process, formal model, state model, software engineering

James Kirby, Jr. is a Member, Technical Staff on the Synthesis Project at the Software Productivity Consortium. Prior to the Consortium he spent three years on the Faculty Technical Staff at The Wang Institute of Graduate Studies and a number of years trying to practice software engineering. He received a Masters of Software Engineering from The Wang Institute. He is interested in software engineering in general, in software design in particular, and in how to foster innovation. Mr. Kirby is a member of the IEEE Computer Society and the ACM.

Robert Chi Tau Lai received the B.A. degree in Architecture in 1981 from Chung-Yuan University, Taiwan, and the M.S. in Architecture in 1985 from the Design Information Processing Laboratory, the Carnegie-Mellon University. He joined the technical staff of the Software Productivity Consortium in 1988, where he is a member of the methodology project. He has worked on object management, Ada task taxonomies, and domain models for software design. His current assignment is to help define the methodology and explore supporting technology for the Consortium's synthesis approach to software development. His principal research interests are design automation methods and tools for software development and architecture design.

Prior to joining the Software Productivity Consortium, Mr. Lai was a visiting research scientist in the school of computer science at Carnegie-Mellon University where he did research on large scale spatial knowledge representation and symbolic computations for geometry and topology. He was a research assistant and manager of the Design Information Processing Laboratory at the Department of Architecture. He also did research on the cognitive process involved in design and on natural language interfaces for computer graphics systems. Mr. Lai is a member of the IEEE Computer Society, ACM, and AAAI.

David M. Weiss received the B.S. degree in Mathematics in 1964 from Union College, and the M.S. in Computer Science in 1974 and the Ph.D. in Computer Science in 1981 from the University of Maryland. He joined the technical staff of the Software Productivity Consortium in 1987, where he has been the leader of the methodology and measurement project. This project has had the responsibility for defining the methodology to be supported by the Consortium's tools. His current assignment is to define the methodology underlying the Consortium's synthesis approach to software development. His principal research interests are in the area of software engineering, particularly in software development methodologies, software design, and software measurement.

Prior to joining the Software Productivity Consortium, Dr. Weiss spent a year at the Office of Technology Assessment, where he was co-author of a technology assessment of the Strategic Defense Initiative. During the 1985-1986 academic year he was a visiting scholar at The Wang Institute. During and prior to his appointment at the Wang Institute, he was a researcher at the Computer Science and Systems Branch of the Naval Research Laboratory (NRL). At NRL he was head of a section that conducted research in software engineering, a member of the Software Cost Reduction project, a consultant to various Navy software development projects, and a participant in a variety of software engineering research projects. He has also worked as a programmer and as a mathematician. Dr. Weiss is a member of the IEEE Computer Society and of the ACM.

1. Introduction

Software design is frequently viewed as an irrational process. Frequent changes in decisions and the large volume of information that must be digested to understand a design prevent the step-by-step, orderly derivation of a design from requirements [8]. One result is that an orderly presentation of the rationale for design decisions is rarely produced. Information that might be used to reconstruct such a derivation is either lost during the process or recorded haphazardly. Recapture of such information after the design is completed is either expensive because of search costs or impossible because the information has been lost. It is only when design is a systematic, repeatable process that there is a chance to systematically record critical information during the process.

Systematizing the design process includes identifying the information needed both during the process and after the process has been completed, and specifying where information of different types must be recorded. Accordingly, it is useful to try to formalize the description of the design process so that one may understand it better, so that one may provide automated support for it, and so that one may provide guidance to software designers concerning what to do at each step and where to record critical information. In this paper we describe an approach to modeling design processes, and give an example of the application of the approach by presenting a formal description of a part of a particular design process. We believe that the same approach can be applied to many design processes.

The Software Productivity Consortium is developing a software design process and tools to support that process. The process we are developing is a variation of the process described in [8]. This paper describes our formal model in order to give an example of the application of formal modeling techniques to the design process.

There are several different approaches that have been proposed for modeling software development. For our purposes we will classify them into procedural models and state models. Procedural models, e.g., [6], assume that the design process can be modeled as parallel sets of sequential events, similar to a set of cooperating sequential processes [3], with each process represented by a program. A design activity, such as producing an interface specification, is defined by a program, which can be executed to simulate the activity. State models, e.g., [5], assume that the process can be characterized as a set of state machines that are executing in parallel, with each state corresponding to a set of activities to be performed by the designer(s). An activity such as producing an interface specification is then one of a number of different states of the process.

Because of the complication in the underlying process, and the amount of backtracking and invention involved, we do not believe that any formal model will both completely and accurately describe software design. Our goal is to be accurate in describing what the work products of design are, what the designers can and cannot be working on, and what the criteria are for completing the work products. We will be incomplete in that we will not attempt to prescribe a total ordering on the events that compose the process. Rather, at points in the process, we will specify that any one of a set of events may next occur. Because state models lend themselves to describing a partial ordering of events better than procedural models, we use a state model. In particular, a state model allows one to define events that permit transitions to predecessor states to represent cases where the designer must go back and redo some or all of a design.

2. The Two Level State Model

We consider a process to be a sequence of decision making activities. Software design is a process of making decisions about (a) what programs should implement the software requirements and (b) what the required properties of those programs are, including properties such as their structure and their interfaces. Information about the decisions made during the design process is captured in artifacts. Example artifacts are a description of the decomposition of a design into a set of components (such as modules, objects, packages, or subroutines) and a specification of the interface of one of the components. To characterize the state of a design we must characterize the state of the artifacts produced during the design process, e.g., whether or not the design decomposition is complete. Just characterizing the state of the artifacts is insufficient to describe a complete design process, however. We must also describe the activities that may be performed on artifacts, the conditions under which those activities are performed, and the roles of the people who may perform them. For example, activities that might be performed on an interface specification are creating it, checking it for completeness and consistency, and conducting a formal review of it. The review will be performed after the specification has been created and analyzed for completeness and consistency. The reviewers may include designers other than the creator, implementors, and quality assurance personnel.

Software design processes vary in many ways, including the way the software is organized into parts, the relations used to define dependencies among the parts, and many other factors. A particular design process may be defined as a sequence of decisions made in different states. A design methodology may be modeled by a set of predefined states,

i.e., it is a prescription for the artifacts to be used, the activities to be performed and their sequencing, and the roles that people play. The definitions of a set of states are the interpretation of the methodology. A designer using the methodology will proceed by following the activities prescribed by the states to produce the prescribed artifacts in the prescribed order. Figure 1 illustrates the elements from which we compose a design model; it is derived from [9]. Arrows in the figure indicate a relation between the elements that they connect, e.g., a role is composed of activities, an activity has subactivities, and an artifact has subartifacts.

The work products of any design process, typically documents, are design artifacts. Most design artifacts are composites or aggregates, i.e., they are composed of a number of parts, each of which is also a design artifact. At some level of organization, the parts may be considered elementary, i.e., for the purposes of design, they are not further decomposable. As an example, Figure 2 is an excerpt from an interface specification for an information hiding module. It is composed of sections, such as a table of operations, each of which is itself an artifact. An entry in the table describing an operation is an elementary artifact.

Our goal in identifying artifacts as part of a design method is to find artifacts whose state can be easily assessed. Furthermore, we would like to be able to associate a rationale with an artifact. Whereas the artifact represents decisions that have been made, we think of the *rationale* for the decision -- what issues led to the making of the decision, what alternatives were considered, and what was the justification for the alternative chosen -- as being a separate concern from the decision recorded by the artifact. The rationale for the decision should be captured elsewhere [10].

Our state model has two levels. The lower level is based on the states of the artifacts produced during the design process; we call such states *artifact states* (A-states). Because A-states alone are insufficient to describe completely the design process, we augment them with descriptions of activities, operations on artifacts, and analyses that can be performed on artifacts within the state, as well as roles of people involved. We call the augmented states in the upper level model *process states* (P-states). A complete description of the state model is presented in a later section.

The two level model allows us to separate the description of the process from the representation used for the artifacts. For example, the upper level model may specify operations to be performed on interfaces to information hiding modules [7] without specifying the representation of such interfaces. (Typical operations to be performed might include creating the specification and baselining the specification.) The specification of the representation may be confined to the lower level model, where the state of the artifact that represents such a module is defined. Figure 3 shows the relationships between P-states, A-states, and the other elements of the design model. The bolded entities and relationships in the figure indicate what has been added to Figure 1. The relationships referenced in Figure 3 will be defined in the following section.

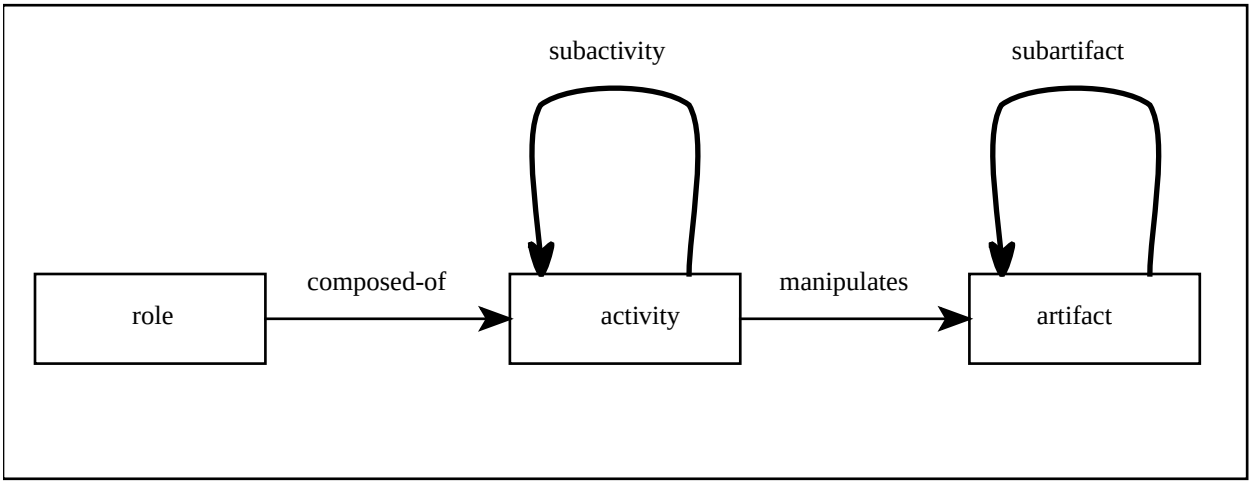


Figure 1 Elements of A Design Model

Sufficiency Requirements: Users of this module need to:

1. . add line. Add a new *line* to the *line list*.
2. . del line. Delete a specific *line* from the *line list*.
3. . chg line. Change the contents of a specific *line* in the *line list*.
4. . ret line. Retrieve the contents of a specific *line* in the *line list*.
5. . trav lines. Traverse the elements in the *line list*.

Feasibility Requirements

1. . store line. There is a mechanism for internally storing and manipulating lines of text.

Operations			
Name	Parameter	Parameter Info.	Undesired Events
LINES	p1:lineno:O	# *lines*	
WORDS	p1:lineno:I p2:wordno:O	*line* # *words* in *line*	LINE_LIMIT_EXCEEDED
CHAR	p1:lineno:I p2:wordno:I p3:charno:O p4:char:I	*line* *word* *character*	LINE_LIMIT_EXCEEDED WORD_LIMIT_EXCEEDED CHAR_LIMIT_EXCEEDED
CHARS	p1:lineno:I p2:wordno:I p3:charno:O	*line* *word*	LINE_LIMIT_EXCEEDED WORD_LIMIT_EXCEEDED
SETCHAR	p1:lineno:I p2:wordno:I p3:charno:O p4:char:I	*line* *word* *character*	LINE_LIMIT_EXCEEDED WORD_LIMIT_EXCEEDED CHAR_LIMIT_EXCEEDED
DELINE	p1:lineno:I	*line*	LINE_LIMIT_EXCEEDED
DELWRD	p1:lineno:I p2:wordno:O	*line*	LINE_LIMIT_EXCEEDED

Effects	
Operation	Description of Effect
SETCHAR	if lineno > 'lines' then LINES := 'LINES' + 1 if wordno > 'WORDS(lineno)' then WORDS(lineno) := 'WORDS(lineno)' + 1 CHARS(lineno,wordno) := 'CHARS(lineno,wordno)' + 1 CHAR(lineno,wordno,charno) := char
DELINE	LINES := 'LINES' -1 For all r greater than or equal to lineno For all wordno For all charno WORDS(r) := 'WORDS(r+1)' CHARS(r,wordno) := 'CHARS(r+1,wordno)' CHAR(r,wordno,charno) := 'CHAR(r+1,wordno,charno)'
DELWRD	WORDS(lineno) := 'WORDS(lineno)' - 1 For all r greater than or equal to wordno For all charno CHARS(lineno,r) := 'CHARS(lineno,r+1)' CHAR(lineno,r,charno) := 'CHAR(lineno,r+1,charno)'

Figure 2 Excerpt from Line Storage Module Interface Specification

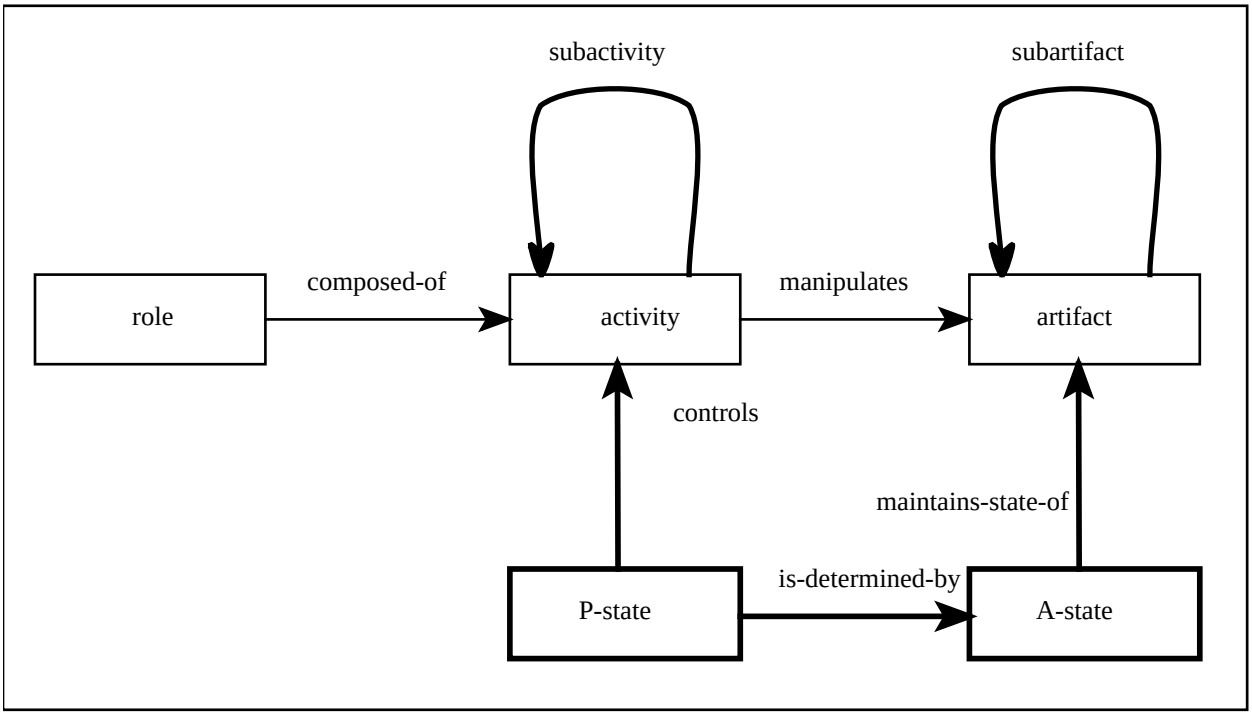


Figure 3
Relationships Defining the Design Model

For different methods, the design artifacts used are different. For example, in our design method, specifications of information hiding modules ([1], [2]) play a key part. We consider an activity to be the work that goes into producing or manipulating a design artifact, i.e., each activity is associated with a particular artifact. (Activities whose design artifacts are composite have subactivities, each associated with a component artifact.) Accordingly, our model is specialized to our design artifacts.

Whether an activity is performable or not depends upon the state of the design artifacts. At any point in time, the set of performable activities represents the choice of artifacts on which the designer may work. Those that are not performable represent the artifacts on which he may not work. Our model prescribes a permissive ordering on activities (and on the work of the designer) by specifying which activities are performable and which are not at any point. By permissive we mean that the designer is free to choose from among the set of performable activities those to pursue. We specify a permissive ordering because we want to support a realistic process. The model captures the information needed to produce a rational description of the work products of an opportunistic process [4].

In addition to specifying artifacts and activities, we also specify the roles played by people involved in design. As with artifacts and activities, different design processes will specify different roles, such as designer, reviewer, implementor, and manager. The roles are defined by the activities in which they may participate.

By augmenting state descriptions with activities and other process-related information, we are able to analyze the needs of the designer at any point in the process. The goal of the analysis is to allow us to present to the designer the information that he needs at the time that he needs it. It also helps us focus on the concerns of what guidance to give to the designer and what analyses can be performed on the design at any time. Finally, it helps us separate the concern of what information should be presented to the designer from how the information should be presented to him.

3. The Artifact State Model

In this section we describe the A-state model for the Consortium design method. We begin by describing the design artifacts. Then we define the A-states that we use to specify the states of design artifacts. Finally, we provide examples of applying A-states to an elementary artifact (i.e., one that contains no other artifacts) and to a composite artifact (i.e., one that is composed of other artifacts).

3.1. The Design Artifacts

The design artifacts for the Consortium design method are listed in . Indentation and decimal numbering are used to indicate which artifacts are aggregations of other artifacts. From the figure we see that the design is an aggregation of four types of artifacts: information hiding structure, dependency structure, process structure, and abstract interface. The first three types of design artifacts are descriptions of design structures. The fourth is a specification of the interface to a component of the design, based on the abstract interface described in [1], [2], and [8]. Figure 2 is an excerpt from such an interface.

The information hiding structure describes decisions about what can and cannot be easily changed. It describes the decomposition of our software into components called *modules*. The dependency structure describes decisions about what artifacts are affected by changes to other artifacts. This information is recorded by the *depends on* relation. If for design artifacts *A* and *B*, (*A*, *B*) is in the relation then we say that *A* depends on *B*; *A* can be present and correct if and only if *B* is also present and correct. *A*, then, is affected by changes to *B*. A part of the dependency structure describes which design requirements depend on other design requirements. An excerpt of the Keyword In Context Index (KWIC) design requirements dependency structure is illustrated in (our examples are taken from a design for the KWIC described in [7]). An arrow from one design requirement to another indicates that the former depends on the latter. The process structure describes the decomposition of the executing software into a set of sequential programs.

The information hiding structure is an aggregation of modules and the is-composed-of relation. Each module is an aggregation of a name and a changeable decision. One can think of the changeable decision as defining the responsibility of the module: the module is responsible for encapsulating the changeable decision. In the KWIC example, the changeable decision of the Line Storage Module is how to store, maintain, and access a list of lines. The module hides the data structures and algorithms used to store, maintain, and access them. The abstract interface for the module provides programs (see Figure 2) that other programs may use to manipulate a list.

The is-composed-of relation, a relation among the modules, defines a tree. A module in the tree is composed of its children. The modules at the leaves of the tree are work assignments for individual programmers or for small teams of programmers. The assignment is to design and implement the module. To make the assignment precise and to identify clearly the decision encapsulated by the module, the programmer produces an interface specification for the module. The module's implementation must satisfy the specification. The interface specified is an abstraction of the changeable decision. For that reason, the interface is known as an abstract interface. Line Storage (see Figure 2) offers to its users an abstraction of a list of lines via its abstract interface.

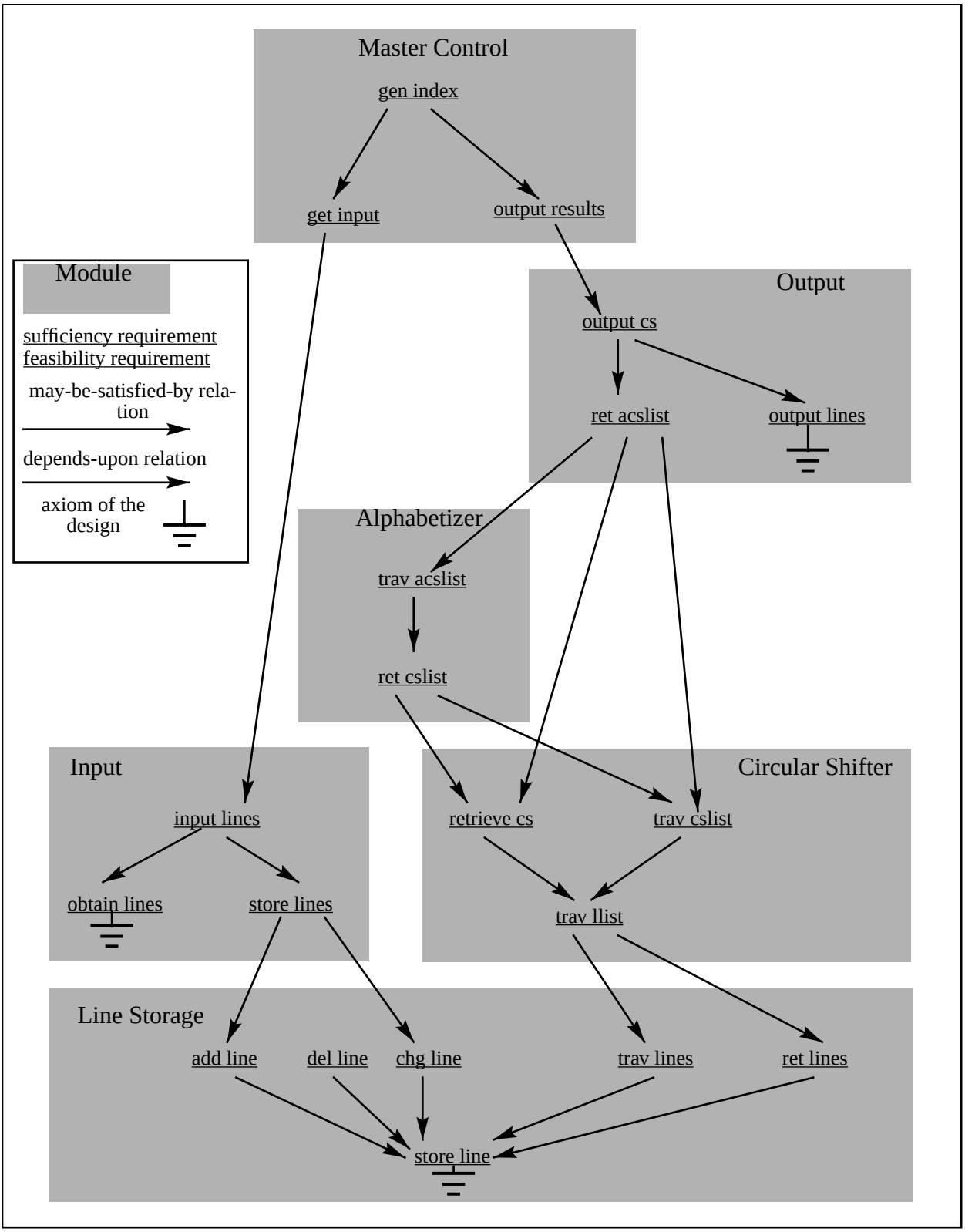
Recall that design artifacts record decisions made by the designer. The abstract interface records decisions about what is visible to users of the module, what users can depend upon, and what might change. Figure 4 lists the artifacts that are components of the abstract interface and the design decisions that each records. Figure 5 provides examples of some of these artifacts taken from Line Storage. As shown in , some of these components of abstract interface are also aggregations of other artifacts. For example, in Figure 5 the operation has subartifacts that are specifications and descriptions of two parameters (*p1* and *p2*) and a reference to an undesired event (*LINE_LIMIT_EXCEEDED*).

We augment the model presented in Figure 3 with design rationale [10]. The augmented model is shown in Figure 6. What has been added to Figure 3 is bolded. The relations in Figure 6 are defined in Figure 7. As part of the design and review activities, issues about artifacts are created and commented on. An issue raises a question about the appropriateness of some aspect of an artifact. For example, a reviewer of the dependency structure in may raise an issue about sufficiency requirement del line, "This sufficiency requirement is not used, can we delete it?" Others might respond to the issue with *yes* and *no* positions, giving arguments, respectively, that "Eliminating del line will make the design simpler." and "Keeping del line will make the design more reusable." Prior to the completion of the design, the designer selects a position for each issue and, if necessary, changes the design to reflect the chosen position. Once an issue is

Design

1. Information Hiding Structure
    1.1. Module
        1.1.1. Name
        1.1.2. Changeable Decision
    1.2. Is-Composed-of Relation
2. Dependency Structure
    2.1. Design Requirements Dependency Structure
        2.1.1. Is-Implemented-by Structure
        2.1.2. Depends-Upon Structure
        2.1.3. Operation-Depends-Upon Structure
        2.1.4. May-be-Satisfied-by Structure
    2.2. Uses Structure
3. Process Structure
4. Abstract Interface
    4.1. Name
    4.2. Sufficiency Requirement
    4.3. Feasibility Requirement
    4.4. Operation
        4.4.1. Name
        4.4.2. Parameter
            4.4.2.1. Name
            4.4.2.2. Type Reference
            4.4.2.3. Mode
            4.4.2.4. Description
        4.4.3. Undesired Event Reference
    4.5. Effect
    4.6. Is-Implemented-by Matrix
    4.7. Depends-Upon Matrix
    4.8. Anticipated Changes
    4.9. Anticipated Subsets
    4.10. Performance Specification
    4.11. Accuracy Specification
    4.12. Type Glossary
        4.12.1. Type Definition
    4.13. Undesired Event Glossary
        4.13.1. Undesired Event Definition
    4.14. Technical Term Glossary
        4.14.1. Technical Term Definition

Design Artifacts
Figure4



Example Design Requirements Dependency Structure
Figure5

created it is considered to be *open* until the designer selects a position and brings the design into conformance with the position.

Figure 4 : Decisions Recorded by Components of Abstract Interface	
Artifact	Decisions Recorded
sufficiency requirement	What is required by users of the abstract interface
feasibility requirement	What implementors of the abstract interface may assume
operation	How the design requirements may be satisfied by a set of operations
effect	What is the behavior of the operations
is-implemented-by matrix	Which operations implement which sufficiency requirements
depends-upon matrix	Which sufficiency requirements depend upon which feasibility requirements
anticipated change	How the abstract interface may change
anticipated subset	What subsets of the abstract interface will be useful
performance specification	What is the required time performance of operations on the abstract interface and what are the space requirements of the module
accuracy specification	What is the required accuracy of output values provided by operations on the abstract interface
undesired event glossary	What classes of abnormal behavior the module will detect
type glossary	What types are required by operations on the abstract interface
technical term glossary	What are the definitions of terms used in specifying the abstract interface

Figure 5 : Examples of Components of Abstract Interface (from Figure 2)			
Artifact	Example		
sufficiency requirement		add line.	Users of this module need to add a *line* to the *linelist*.
feasibility requirement		store line.	There is a mechanism for storing, retrieving, and manipulating lines of text.
operation	words	ParameterDescription p1:lineno:I*line* p2:wordno:O# *words* in *line*	Undesired Event LINE_LIMIT_EXCEEDED
is-implemented-by matrix	(add line, setchar)		
depends-upon matrix	(add line, store line)		
anticipated change	We may want to allow the user to maintain more than one *linelist* concurrently.		
anticipated subset	BASIC = (add line ret line)		
undesired event glossary	LINE_LIMIT_EXCEEDED.p1 is not a valid lineno.		
type glossary		lineno.	Handle for a *line*.
technical term glossary		*linelist*.	An ordered set of *lines*.

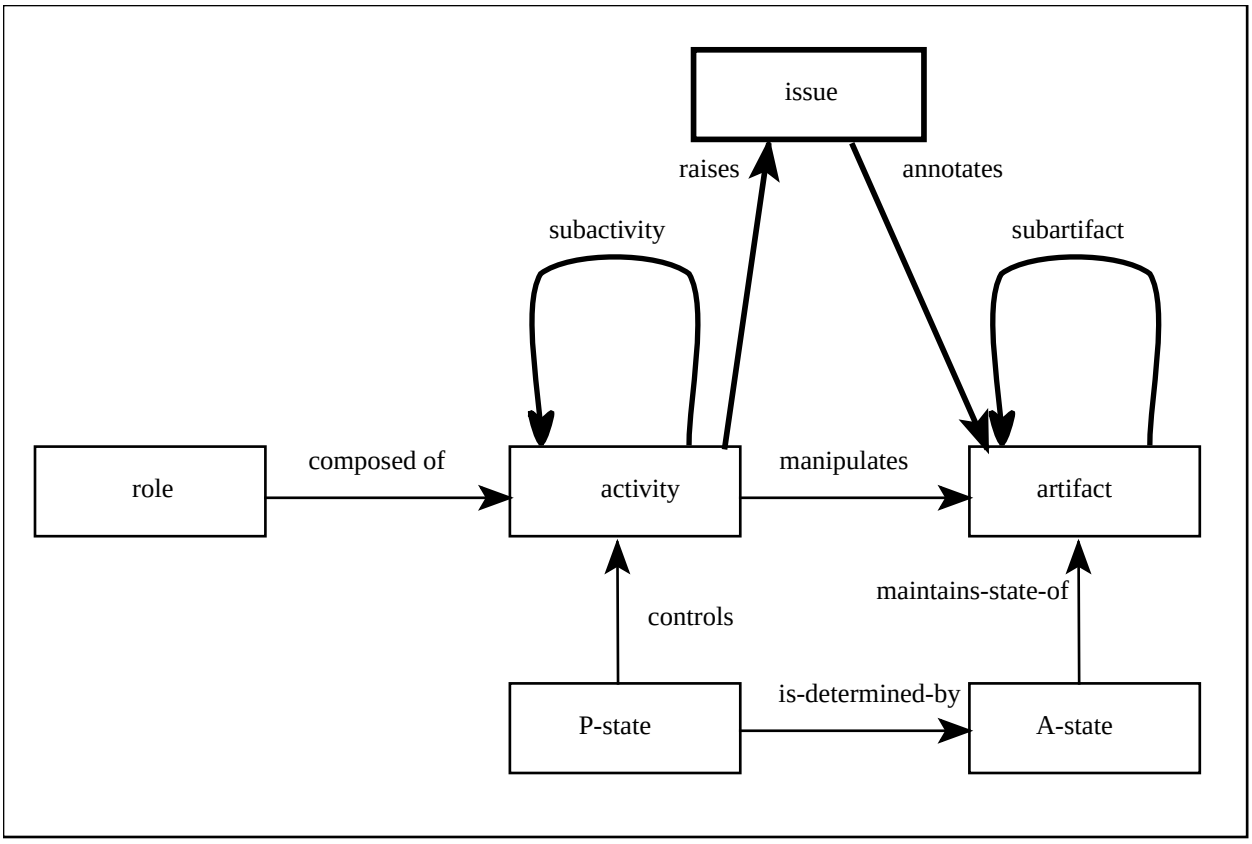


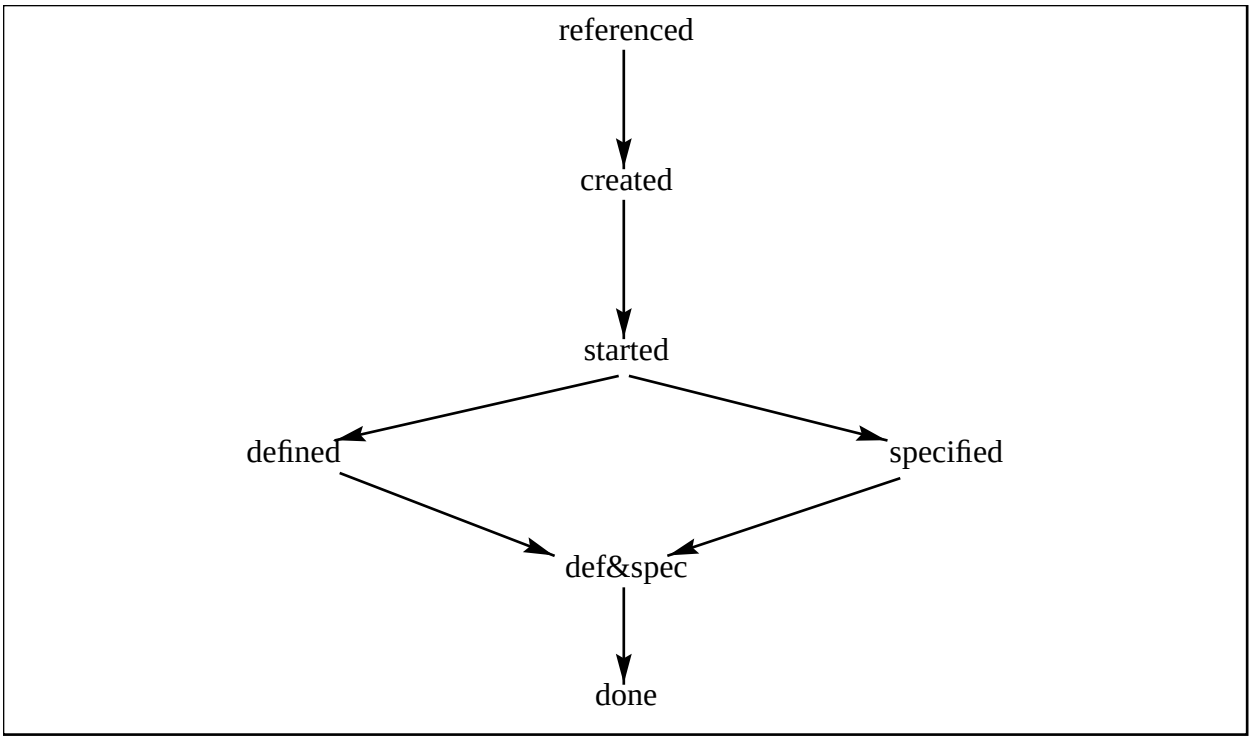
Figure 6
Relationships Defining the Design Model

Figure 7 : Definitions of Design Model Relationships			
Relation	Domain	Range	Definition
annotates	issue	artifact	The issue raises a question about the design of the artifact. (many to many)
composed of	role	activity	Someone playing the role may perform the activity. (many to many)
controls	P-state	activity	Whether the activity is performable is determined by the P-state. (many to many)
is determined by	P-state	A-state	Whether the design process is in the P-state is determined by a relation on the A-state. (many to many)
maintains state of	A-state	artifact	The state records information that defines the current state of the artifact.
manipulates	activity	artifact	The artifact is an output of the activity. (one to many)
raises	activity	issue	The issue is an output of the activity. (one to many)
subactivity	activity	activity	The activity in the domain is composed of the activities in the range. (one to many)
subartifact	artifact	artifact	The aggregate artifact in the domain is composed of the artifacts in the range. (one to many)

Figure 8 : Design Artifact States	
State	Definition
referenced	There is a reference to the artifact, but there is neither a name nor any work associated with it.
created	There is either a name or work associated with the artifact.
started	There are both a name and work associated with the artifact.
defined	There are both a name and work associated with the artifact and the work either depends upon no other artifacts or any artifacts that it depends upon are defined.
specified	There are both a name and a specification for the artifact and the specification either depends upon no other artifacts or any artifacts that it depends upon are specified.
def&spec	There are a name, a specification, and other work associated with the artifact. The specification either depends upon no other artifacts or any artifacts that it depends upon are specified. The other work either depends upon no other artifacts or any artifacts that it depends upon are defined.
done	There are a name, a specification and/or other work associated with the artifact. If there is a specification it either depends upon no other artifacts or any artifacts that it depends upon are done. If there is other work it either depends upon no other artifacts or any artifacts that it depends upon are done. The artifact has been reviewed and there are no open issues associated with it.

3.2. The Artifact States

We have defined a set of seven A-states that formalize the state of completeness of each of our design artifacts. The A-states are listed and informally defined in Figure 8. Every one of our design artifacts is in one of these A-states. The A-state of an artifact describes the completeness of the artifact. We define a partial ordering relation on the A-states (illustrated in). In the figure, an arrow from *A* to *B* indicates that an artifact in A-state *B* is more complete than an artifact in A-state *A*. We think of the A-states as defining a state machine for each design artifact. Actions performed by the designers and other participants in the design process cause the state machines for the artifacts to transition from one A-state to another. We use the A-states, a formalization of the completeness of the artifacts, to guide us in completing



Partial Ordering of Artifact States
Figure11

the design. We provide more precise and formal definitions of artifact state when we discuss the individual design artifacts.

Because we have defined a partial ordering on the A-states, we can use the relational operators *greater than*, *less than*, *greater than or equal*, and *less than or equal* on A-states. For example, for A-states *A* and *B*,

$$B > A$$

means that an artifact in A-state *B* is more complete than an artifact in A-state *A*.

The state machines for some of the artifacts will not include all of the A-states, because those artifacts can never be in some of the states. For example, we write sufficiency requirements in natural language. The sufficiency requirement add line from the Line Storage Module is "Users of this module need to add a *line* to the *linelist*." Because it does not make sense to say that the sufficiency requirement has been specified, the state machine for a sufficiency requirement does not include the A-states specified and def&spec.

In the preceding discussion of artifact state we have focused on the set of A-states we have chosen for our design process. We would like to distinguish between what is true of A-states in general and what is true of the set of A-states that we have defined to model our design process. For any design process that one wishes to define using our two level state model, one specifies a state machine for each artifact in the process. For a particular process, one may use the seven A-states that we have defined or one may want to define another set of A-states that are better suited to the process. The A-states that make up the state machine for each artifact may be the same for all of the artifacts (as was the case with our process) or they may be different. It is convenient and leads to a simpler model if all of the state machines use the same set of A-states, but for some processes it may not be desirable or possible to use the same set of A-states for every artifact.

3.3. The States of Design Artifacts

In the following paragraphs we describe how to determine the state of an artifact. First, we discuss how the state of an elementary design artifact is determined. Then we discuss how to determine the state of an aggregate design artifact.

The state of an elementary design artifact, one that is not an aggregation, is determined by examining the artifact itself, whether there are any open issues that annotate it, and the state of artifacts on which the artifact depends. Figure 9 defines the A-states (and the A-state machine) for a simple artifact, a sufficiency requirement. Predicates used in Figure 9 are defined in Figure 10. A small example of how a sufficiency requirement transitions among the A-states is illustrated in Figure 11.

A sufficiency requirement is a component of an abstract interface. It is a concise statement of what is required by users of the abstract interface. A sufficiency requirement may depend upon other artifacts, which may affect its state. For example, if a sufficiency requirement references a technical term that is not at least in the defined state, the sufficiency requirement can not be in the defined state. Informally, the sufficiency requirement can't be considered to be defined unless all the technical terms it uses are defined. As shown in Figure 9, a sufficiency requirement is in the defined A-state if it has a name, is not empty, and either refers to no other artifacts or any artifacts to which it does refer are defined. A sufficiency requirement is in the done state if it satisfies the conditions required to be defined, has been subjected to a technical review, has not been changed since the review, has no open issues associated with it (the output of a review), and either refers to no other artifacts or refers only to artifacts that are done.

Figure 11 illustrates how a sufficiency requirement and two technical term definitions upon which it depends transition through A-states. In the figure, underlining indicates a sufficiency requirement name (add line) and italics indicates a technical term name (*linelist* and *line*). Note that when add line is first defined it is in the defined A-state because it refers to no other artifacts. When the definition is modified to include a reference to the undefined *linelist*, add line transitions to the started A-state.

Figure 9 : State Condition Table for a Sufficiency Requirement (sr)

State	Condition
referenced	$(\exists x)(Refer(x, sr) \wedge \sim Name(sr) \wedge Empty(sr)$
created	$Name(sr) \vee \sim Empty(sr)$
started	$Name(sr) \wedge \sim Empty(sr)$
Defined	$Name(sr) \wedge \sim Empty(sr) \wedge$ $(\sim(\exists y)Refer(sr, y) \vee (\forall y)(Refer(sr, y) \Rightarrow StateOf(y) >$ started))
Done	$Name(sr) \wedge \sim Empty(sr) \wedge Rvw(sr) \wedge \sim OI(sr) \wedge$ $(\sim(\exists y)Refer(sr, y) \vee (\forall y)(Refer(sr, y) \Rightarrow StateOf(y) =$ Done))

Figure 10 : Definitions of Predicates

Predicate	Definition
Empty(x)	There is no work associated with the object x. The object is empty.
Name(x)	There is a name associated with object x.
OI(x)	There are open issues associated with the object x.
Refer(x, y)	Object x references object y.
Rvw(x)	The object x has not changed since it was subjected to a technical review.

Figure 11 : Artifact State Example for Sufficiency Requirement			
Action	State of Artifact Following Action		
	Add Line	*Linelist*	*Line*
1. Sufficiency requirement created and named: add line.	created		
2. Add line defined: "Users of this module need to add to the linelist."	defined		
3. Add line redefined: "Users of this module need to add to the *linelist*."	started	referenced	
4. Technical term definition *linelist* created, named, and defined: "An ordered set of lines."	defined	defined	
5. Add line redefined: "Users of this module need to add a *line* to the *linelist*."	started		referenced
6. *Linelist* redefined: "An ordered set of *lines*."		started	
7. Technical term definition *line* created, named, and defined: "An ordered set of words."	defined	defined	defined
8. Add line, *linelist*, and *line* are reviewed and there are no open issues associated with them.	done	done	done

Figure 12 : State Condition Table for Abstract Interface (ai)	
State	Condition
referenced	(∃x)(Refer(x, ai) ∧ ~(∃x)(comp(x, ai) ∧ StateOf(x) > referenced)
created	(∃x)(comp(x, ai) ∧ StateOf(x) > referenced) ∧ (∃x)(comp(x, ai) ∧ StateOf(x) <= created)
started	(∀x)(comp(x, ai) ⇒ StateOf(x) >= started) ∧ (∃x)(comp(x, ai) ∧ StateOf(x) = started)
Defined	(∀x)(comp(x, ai) ⇒ StateOf(x) > started) ∧ (∃x)(comp(x, ai) ∧ StateOf(x) = defined)
specified	(∀x)(comp(x, ai) ⇒ StateOf(x) >= specified) ∧ (∃x)(comp(x, ai) ∧ StateOf(x) = specified)
def&spec	(∀x)(comp(x, ai) ⇒ StateOf(x) >= def&spec) ∧ (∃x)(comp(x, ai) ∧ StateOf(x) = def&spec)
Done	(∀x)(comp(x, ai) ⇒ StateOf(x) = done) ∧ Rvw(ai) ∧ ~OI(ai)

Figure 13 : Definitions of Predicates	
Predicate	Definition
Comp(x, y)	Artifact x is a component of aggregate artifact y.
OI(x)	There are open issues associated with the object x.
Refer(x, y)	Object x references object y.
Rvw(x)	The object x has not changed since it was subjected to a technical review.

Figure 14 : Artifact State Example for Line Storage Abstract Interface				
Action	State of Artifact Following Action			
	Abstract Interface	Add Line	*Linelist*	*Line*
	done			
Sufficiency requirement created and named: add line.	created	created		
Add line defined: "Users of this module need to add to the linelist."	def&spec	defined		
Add line redefined: "Users of this module need to add to the *linelist*."	created	created	referenced	
Technical term definition *linelist* created, named, and defined: "An ordered set of lines."	def&SPEC	defined	defined	
Add line redefined: "Users of this module need to add a *line* to the *linelist*."	created	created		referenced
Linelist redefined: "An ordered set of *lines*."			created	
Technical term definition *line* created, named, and defined: "An ordered set of words."	def&spec	defined	defined	defined
Add line, *linelist*, and *line* are reviewed and there are no open issues associated with them.	done	done	done	done

The state of an aggregate design artifact is determined by whether there are any open issues that annotate it, the state of its components, and the state of artifacts upon which the aggregate artifact and its components depend. The state of the aggregate artifact is bound by the minimum of the states (determined by the partial ordering illustrated in) of those artifacts that compose it and those artifacts upon which the aggregate artifact and its components depend. For example, the A-state of an aggregate artifact can be no greater than created if a subartifact is referenced. The aggregate artifact can be no greater than defined if a subartifact is defined. We chose the minimum because we want to point the designer toward what might be done next to move the design toward completion.

Figure 12 defines the A-states for the aggregate artifact abstract interface. Predicates used are defined in Figure 13. Figure 14 illustrates how an abstract interface transitions through A-states as the A-states of its components change. In the first row of the table, the abstract interface is in the done A-state, because all of its components are done, the abstract interface has been reviewed, and there are no open issues associated with it. In the following rows we see how the A-state of the abstract interface changes as a new sufficiency requirement and two new technical terms are added to the abstract interface and changed. In the last row of the table we see that a successful review (*i.e.*, one that results in no open issues) will move the new sufficiency requirement, the two new technical terms, and the abstract interface to the done A-state.

3.4. Summary

In this section we have discussed A-states and the A-state machine associated with each design artifact. We have provided examples of A-states, A-state machines, and design artifacts. We have shown how the A-state machines transition as the designer changes the associated artifacts. For our design process, we defined seven A-states that we use to describe the state of each of the artifacts in our process. The seven A-states that we defined are convenient for understanding, managing, and applying the process. We believe that the seven states that we have chosen correspond well to intuitive notions of completeness. The small number of states used throughout the description makes for a simpler, more understandable model. We can formally define the A-states for each artifact which will reduce the ambiguity in trying to assess the state of completeness of a design.

4. The Process State Model

4.1. Process States

The components of a P-state are shown in Figure 15. A P-state is characterized in terms of the A-states of the artifacts associated with the P-state. Consequently, the specification of a P-state includes a list of the artifacts associated with it. Figure 16 is an example P-state, named Abstract_Interface_Design. It describes a state in which work may progress on designing an abstract interface. Accordingly, it is characterized in terms of the artifact *design.abstract_interface*. (The notation for naming the artifact shows where in the design artifacts hierarchy (Figure 4) the artifact belongs.)

In addition to artifacts associated with the P-state, its specification also describes the precondition for state entry, and activities that may be performed in the state. The precondition is the condition that must be true to enter the state. Activities are of two types: operations, which may be used to change the state of an artifact, and analyses, which return information about the design in general, including guidance on design methods, and artifacts in particular.

Figure 15 : Components of Process State (P-State)

Components	Definitions
Name	The name of the state.
Precondition	Predicates on A-states that determine when the process enters the P-state.
Artifacts	A list of artifacts on which work may proceed in the P-state
Activities	A list of operations and analyses that may be performed in the P-state. See Figure 17 and Figure 19 for detailed descriptions of operations and analyses.

As an example, in the P-state Abstract_Interface_Design, the operations Create, Edit, Review, and Baseline may be performed on the abstract interface artifact. Any of these may change the A-state of the artifact, e.g., successful conclusion of the Baseline operation may change the A-state of the abstract interface to Done. (See section 3 for a discussion of the A-states of an abstract interface.) On the other hand, the analyses Consistency, Dependency, Undefined_terms, and Completeness may also be performed on the artifact. These analyses return information about the artifact, e.g., the Completeness analysis will provide information about components of the artifact that are not yet created. The operations and analyses associated with a P-state are described in further detail in Figure 17 and Figure 19, respectively.

Figure 16, Figure 18, and Figure 20 show an example of a P-state. The example is not meant to be complete, but just to illustrate how our process model can be applied.

In specifying predicates in the example, we have made liberal use of functions that return information about the design. For example, the function *state_of* is a state retrieval function that returns the states of design artifacts. For the purposes of this paper, we will assume these functions are primitive, and provide only informal definitions of them as needed in discussing the example. The following sections discuss the components of a P-state in more detail.

4.1.1. Operations

An operation may be specified by a pre_condition, the design artifacts to which the operation may be applied, actions, roles, and post-actions, as shown in Figure 17. Pre_conditions are represented for operations the same as for P-states; post-actions are represented as lists of actions.

An action may be performed when the pre-condition for the operation is true. The action embodies the effect of the operation. Accordingly, the description of an action is a formal specification of the effects of the operation. As in other examples, we have assumed the actions are primitive functions and not defined them here in detail. Since not all design operations are permissible to all the people who participate in design, an operation also has a list of roles of people allowed to perform the operation.

Figure 18 defines the operations for the example P-state Abstract_Interface_Design. The Create operation can only be performed when there is a module with the same name as the abstract interface, and the module is at least in the Created state. Informally, a module must first exist before an abstract interface can be created for it. The Create operation can

be performed by a designer, operates only on an abstract interface, and its effect is defined by the function create_abstract_interface. It has no post-actions associated with it.

Figure 16 : An Example Process State (P-State)	
Components	Values of (Sub)components
Name	Abstract_Interface_Design
Pre_conditions	(state_of(design.module) >= Started) ∧ (state_of(design.module) < Done) ∧ name_of(design.module) = name_of(abstract_interface)
Artifacts	Design.Abstract_Interface
Operations	Create, Edit, Review, Baseline
Analyses	Consistency, Dependency, Undefined_terms, Completeness

Figure 17 : Components of an Operation	
Components	Definitions
pre-condition	Predicates on A-states that determine when the operation is active.
design artifacts	The design artifact manipulated by the operation.
action	The function to be performed by the operation.
roles	A list of roles of people who are allowed to perform this operation.
post-action	A list of actions to be performed after the operation is completed.

Figure 18 : Operations for Abstract_Interface_Design State		
Components	Sub-Components	Values of (Sub)components
create	pre-conditions	((state_of(design.module) >= Started) ∧ name_of(design.module) = name_of(abstract_interface) ∧ (state_of(design.abstract_interface) <= Referenced)))
	design artifacts	design.module.abstract_interface
	actions	create_abstract_interface(design.abstract_interface)
	roles	designer
	post-actions	none
edit	pre-conditions	(state_of(design.abstract_interface) >= Created)
	design artifacts	design.abstract_interface
	actions	edit_abstract_interface(design.abstract_interface)
	roles	designer
	post-actions	none
review	pre-conditions	(state_of(design.abstract_interface) >= Started)
	design artifacts	design.module.abstract_interface
	actions	review_abstract_interface(design.abstract_interface)
	roles	designer, reviewer, manager
	post-actions	none
base_line	pre-conditions	(state_of(list_of_dependent_artifacts_of(design.abstract_interface)) >= DEF&SPEC)
	design artifacts	design.abstract_interface, list_of_dependent_artifacts_of(design.abstract_interface)
	actions	baseline(design.abstract_interface)
	roles	manager
	post-actions	none

Figure 19 Components of Analysis	
Components	Definitions
Design artifact	A list of design artifacts related to this analysis.
Analysis	Specification of the analysis to be performed.
Roles	List of roles of people who may perform the analysis.
Context	The context of the analysis
Trigger	How the analysis can be invoked.
Action	Actions to be performed depending upon the results of the analysis.

Figure 20 : Example Analyses

Name of Analysis	Sub-Components	Values of (Sub)components
consistency	design artifact	design.abstract_interface

	analysis	function ana_abs_int_const(design.module)
		(for module_var in design
		(if use(module_var, design.module)
		then
		for operation_var in module_var
		if (~consistent(design.module.abstract_interface,
		module_var.operaion_var))
		then collect(result,(design.abstract_interface,
		module_var.operation_var))
		if use(design.module, module_var2)
		then for operation_var2 in design.module
		if (~consistent(design.module.operaion_var2,
		module_var2.abstract_interface,))
		then collect(result,
		(module_var2.abstract_interface,
		design.module.operaion_var2)))
		return result)

	context	syntactic
	trigger	@T(user_requests_consistency_analysis) ∨ @T(state_of(design.abstract_interface) = Defined)
	action type	prompt_message(err_info56), write_to_file(file_name_var), mail_manager(message29,managers_list_var), mail_designer(message_22,designer_list_var), mail_reviewer(message_82, reviewer_list_var)
	roles	designer, manager, reviewer
undefined_terms	design artifact	design.abstract_interface
	analysis	function undefined_terms (design.abstract_interface) (for word_var in design.abstract_interface if high_lighted(word_var) & ~in_design.dictionary(word_var, design.module.dictionary) then collect(word_var, result) return result)
	context	textual
	trigger	@T(user_requests_undefined_terms_analysis) ∨ @T(state_of(design.abstract_interface) = Defined)
	action type	prompt message(message44), write_to_file(file_name_var, format61), mail manager(message77, managers_list9), mail_designer(message33, designer_list_var), mail_reviewer(message78, reviewer_list_var)
	roles	designer, manager, reviewer

4.1.2. Analyses

In a P-state, a designer can get information about the current status of the design by invoking an analysis. The information offered by the analysis helps designers make decisions. An analysis is defined by the design artifacts to which it applies, the functions it performs, its types (context type, trigger type, and action type), and roles of the people who may invoke it, as shown in Figure 19. The type of each analysis is determined by the following attributes:

1. Context:
   a representation of the basis for the analysis, such as quality assurance rules, syntactic analysis, or semantic analysis. For example, a consistency analysis of an abstract interface might be based on the syntax of the interface, i.e., the analysis might verify that all sections of the interface existed and that every technical term used in the interface was defined in a technical terms glossary.
2. Trigger:
   a representation of the events that can trigger the analysis, such as a direct request by the user. An event is a point in time at which a certain condition become true, e.g., entry into a P-state, exit from a P-state, at the expiration of a time interval, or when an artifact enters or exits a particular state. The example in Figure 20 uses the notation used in [2] to specify events. For a predicate *p*, we use @T(*p*) to specify an event. @T(*p*) specifies the point in time when *p* goes from false to true.
3. Action Type:
   Actions may be specified as part of the analysis definition for the cases when certain conditions are found to be true from the analysis. Example actions include (*state* is the state of a design artifact):
   a. prompt message (*state, message*) :
      prompt the designer(s) with the *message* when the analysis returns *state*.
   b. save(*state, name*) :
      save the result of analysis in *name*, when the analysis returns *state*.
   c. mail manager(*state, list of user_ID, message*) :
      mail to the manager(s) (*list of user_ID*) the *message* when the analysis returns *state*.
   d. mail designer(*state, message*) :
      mail to the designer(s) the *message* when the analysis returns *state*.
   e. mail reviewer(*state, list of user_ID, message*) :
      mail to the reviewer(s) (*list of user_ID*) the *message* when the analysis returns *state*.

Figure 20 illustrates a set of analyses for the P-state, Abstract_Interface_Design.

4.1.3. Roles

A role defines the responsibilities of jobs associated with software design, such as designer, manager, or reviewer. A person who plays a designated role can determine what activities he may perform by looking at all the operations and analyses permitted for that role over all states. We expect that roles will vary according to the design method used, e.g., a reviewer's role in one design method may be quite different than in another design method.

4.2. The Process Model As A State Machine

Because designers are likely to be working on several aspects of a design concurrently, we model the design process as a concurrent state machine, i.e., a state machine that can be in several states simultaneously. This permits a designer to make progress in one state for a while and then move to another without completing the first. It also permits several designers to work on different parts of the design concurrently. For example, in our design method, definition of a module precedes specification of its abstract interface. Once several modules have been defined, work may proceed on their abstract interfaces concurrently. We model this situation by creating a P-state for each module and for each abstract interface and by requiring as a pre-condition that a module must be defined before work may proceed on its abstract interface, as specified in the example in Figure 16. Our state machine may then be in several of these states at the same time. By offering this capability, the process model can allow designers freedom to act opportunistically, yet still provide a means for guiding them in an orderly way. In addition, it can be used to record an opportunistic design process so that the design can later be presented in a rational way.

Modeling the design process as a concurrent state machine allows us to satisfy several concerns. First, it gives us a precise definition of our design methodology. Designers, reviewers, and others involved in the process have a reference

that specifies what they can do at any stage of the process. Second, it permits a parallel design process, since the model identifies opportunities for different designers in a team to work concurrently and independently. Third, it gives us a specification for a design tool to support the process, i.e., a tool that could keep track of currently active design states, that could record information about the design, and that could provide the necessary operations and analyses for the designer to make progress on the design.

4.3. Questions That Could Be Answered By Our Formal Model

A primary motivation for constructing our model was to answer the following questions at any stage of the design process for software designers using our methods:

1. What do I do next?
2. What output do I produce?.
3. What input do I need?
4. How do I know when I'm done?
5. Who has responsibility what?

To the extent that the model provides capabilities for performing design activities, such as operations and analyses, it also answers the question,

6. How do I do it?

5. Constructing A Model

The previous sections have discussed the elements of the formal model and shown examples of A-states and P-states. In this section we describe an approach to constructing a model for a particular process. To complete such a model requires that the modeler have a clear idea of the artifacts and activities that are used in the process that he is modeling. While the modeler may not have a sufficiently clear view of the process at the outset, one intent of our modeling process is to help him obtain such a view. The following steps form a bottom up approach to constructing a model that may help clarify the model under construction. As with most processes associated with software development, some back-tracking will be needed in following this process.

1. Define a set of design artifacts according to the methodology to be used.
2. Identify dependency and composition relations among the artifacts.
3. Use the A-state model to identify a set of states for each design artifact.
4. Identify a set of operations that can be performed on each design artifact.
5. Identify pre-conditions and post actions for each operation.
6. Identify a set of analyses that software developers may need for making decisions in conjunction with each operation.
7. Identify guidelines and methods for each operation, and where possible, translate them to analyses.
8. Group the operations according to those that need to be done about the same time.
9. Collect the analyses, guidelines, and operations together according to the grouping in step 8. to form a P-state.
10. Identify pre-conditions for each P-state.

The result of this process is a set of A-states and P-states that have a flat structure. For particularly complicated models, it may be useful to introduce a hierarchical structure into the set of P-states. For example, if a complete software development process were being modeled, then design might be represented at the top level of the model as a single P-state. The design P-state might itself then be decomposed into a hierarchy of P-states. For such cases, we add an additional step to the modeling process:

11. Identify super-P-states that are groups of P-states to build a hierarchical machine.

6. Using the Model to Build Design Tools

6.1. Building Design Tools

A two-level model of a design process can be viewed both as a specification for the process and as a specification for tools to be used to support the process. The model specifies both the data on which the tools must operate (i.e., the set of design artifacts and their interdependencies), the operations to be performed by the tools, and the effects of those operations (i.e., the state changes caused by invoking the tools). Accordingly, a particular model may be used to explain the process to software developers who will use it, to tool developers who will build tools to support it, or to tool integrators who will integrate a set of existing tools to support the process.

We have deliberately left the type of specification to be used in describing operations and analyses to the discretion of the modeler. For different purposes and audiences, different types will be appropriate. For the purpose of describing the process to software developers, the modeler may use an informal prose specification or a formal specification that describes effects in terms of state changes on composite artifacts. As an example of the latter, the effect of using an editor to create an abstract interface might be specified as

state_of(abstract_interface) := Created

For the purpose of describing the services offered by the editor to the software developer, the modeler may provide a reference to the user's manual for the editor. For the purpose of describing to a tool builder the interface provided by the editor, the modeler may reference an interface specification for the editor. For the purpose of specifying the tool to be built, the modeler may reference a requirements specification for the editor. Any combination of such references and specifications may be used by the modeler, as appropriate.

In particular, the implementor of the tool must implement actions that are implicitly specified by the model. For example, in cases where there are dependencies among design artifacts, a state change in one artifact may cause state changes in other artifacts, as shown in Figure 11. Furthermore, there may be informal parts of the model that are not well-captured by the formal model, e.g., specifying methods for resolving contending requests to update the same design artifacts by several different designers. Such requests may arise directly, e.g., when several designers are simultaneously attempting to manipulate the same design artifact, or indirectly, when several designers simultaneously change the states of dependent design artifacts.

One way to make implicit actions more explicit for the purposes of implementation, or to specify conflict resolutions, is to provide a way of specifying actions to be taken on the occurrence of certain types of events. One useful such class is post-actions accompanying state exits, particularly P-state exits. For example, on leaving the P-state Abstract_Interface_Design, the following post-action might be taken.

if (state_of(abstract_interface) = done) ==> notify(reviewers_of(abstract_interface))

6.2. Questions That Could Be Answered By A Tool

A tool developed to support our model would also be able to answer questions such as:

1. What is the difference between version X and version Y of the software?
2. What modifications have been made since a design artifact reached a particular state, or since a particular P-state was last visited?
3. What modules are affected by this or that particular modification or batches of changes or modifications? Who modified them, and for what reason?
4. What remains to be done to complete a particular state?
5. What are the states of various design artifacts?

6.3. Integrating Design Tools

The process model can be used as an integration model to integrate different design tools to support a design process. This can be done in the following steps:

1. For each operation and analysis in each P-state, select a set of commercially-available tools that can be used to perform the operations and analyses,
2. Design the interfaces that permit tools access to design artifacts and their states,

3. Implement actions as invocations of an existing commercially-available tool using the interfaces.

Using this procedure, the model becomes an instrument for designing a common tool environment.

7. The Benefits of Our Method of Modeling the Design Process

We believe that there are several benefits to our method for developing formal models of software design processes. The method can be applied to a wide range of processes and methods. The formal models produced help to understand the process, to manage it, and to apply the process in developing software designs.

The method can be applied to any design method or process that has well-defined design artifacts and well-defined composition and dependency relations among the artifacts. Well-defined artifacts and relations allow one to formalize the artifact states upon which the formalization of the process states and of the process depend. Well-defined artifacts allow one to specify what artifacts record the design. Well-defined artifacts and well-defined relations among them allow one to specify how to determine the extent of completeness of the design. For artifacts whose completeness can't be formally specified, we use technical reviews and tracking of the issues that result from them to specify completeness.

The application of our method will result in process models that are relatively simple and clear, that can describe realistic, opportunistic processes and traditional waterfall processes, and that can provide useful guidance to the designer applying the process. These characteristics make the models useful to those trying to understand, manage, and follow the process. The models tend to be simple because our two-level approach allows us to separate concerns. For example, we can use the model to think about what is the state of the design without being concerned with how to determine the state. We have shown how we model part of an opportunistic design process. We model a waterfall process by developing an ordering of the process states and by specifying in the precondition of each P-state that it can not be entered until previous P-states have been exited. We can provide the designer with guidance on what to do next based upon the state of the design. Such guidance could be usefully provided either automatically online or in appropriately indexed hard-copy.

Acknowledgements

We are grateful for thoughtful reviews performed by Paul Clements and Jim O'Connor. The example abstract interface was provided by Jeff Facemire and Jim O'Connor.

8. References

[1] Britton, K.H., R. A. Parker, and D.L. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules," Proc. 5ICSE, pp. 195-204, 1981.

[2] Clements, P.C., R.A. Parker, D.L. Parnas, J.E. Shore, and K.H. Britton. A Standard Organization for Specifying Abstract Interfaces, NRL Report 8815, June 14, 1984.

[3] Dijkstra, E. W. "Co-operating Sequential Processes," Programming Languages, ed. F. Genuys, New York: Academic Press, pp. 43-112

[4] Guindon, R. A Framework for Building Software Development Environments: System Design as Ill-structured Problems and as an Opportunistic Process, MCC Technical Report STP-298-88, Septmeber 18, 1988.

[5] Kellner, M. "Representation Formalisms for Software Process Modeling," Proc. 4 Intntl. Software Process Workshop

[6] Osterweil, L. "Software Processes Are Software Too," Proc. 9ICSE, March 1987

[7] Parnas, D.L. "On the Criteria to be Used in Decomposing a System into Modules," Comm. ACM, Vol. 15. No. 12. pp. 1053-1058.

[8] Parnas, D.L. and P.C. Clements. "A Rational Design Process: How and Why to Fake It, "IEEE Trans. on Software Engineering, February, 1986.

[9] Potts, C. "A Generic Model for Representing Design Methods," Proc. 11ICSE, May, 1989

[10] Potts, C., and G. Bruns. "Recording the Reasons for Design Decisions," Proc. 10ICSE, April 1988.